

1 KEKER & VAN NEST, LLP
DARALYN J. DURIE - #169825
2 EUGENE M. PAIGE - #202849
RYAN M. KENT - #220441
3 SONALI D. MAITRA - #254896
710 Sansome Street
4 San Francisco, CA 94111-1704
Telephone: (415) 391-5400
5 Facsimile: (415) 397-7188

6 Attorneys for Defendant
WELLS FARGO BANK, N.A.
7
8

9 UNITED STATES DISTRICT COURT
10 NORTHERN DISTRICT OF CALIFORNIA
11 SAN FRANCISCO DIVISION

12 PHOENIX SOLUTIONS, INC., a California
13 corporation,

14 Plaintiff,

15 v.

16 WELLS FARGO BANK, N.A., a Delaware
17 corporation

18 Defendant.
19
20

Case No. CV 08-0863 MHP

**DECLARATION OF R. DOUGLAS
SHARP IN SUPPORT OF DEFENDANTS'
MOTION FOR SUMMARY JUDGMENT
OF INVALIDITY**

Date: November 10, 2008
Time: 2:00 p.m.
Dept: Courtroom 15, 28th Floor
Judge: Hon. Marilyn Hall Patel

Date Comp. Filed: February 8, 2008

Trial Date: TBD

21 I, R. Douglas Sharp, declare as follows:

22 1. I have personal knowledge of the facts stated in this declaration, and could and
23 would testify to these facts under oath if called upon to do so.

24 **A. My Employment History**

25 2. I am currently **Nuance's** Vice President of Enterprise Engineering, responsible for
26 leading research and development including core network ASR technologies, OEM ASR,
27 Verification, and NLU engines, the **Nuance** Voice Platform, application framework, tools, and
28 packaged applications.

1 3. Prior to the merger between **Nuance** and ScanSoft, I was Executive Vice
2 President of Engineering for **Nuance** Communications, responsible for leading all of **Nuance's**
3 R&D.

4 4. Prior to joining **Nuance** in 1998, I led the collaboration between AT&T product,
5 development, and research organizations to build the AT&T Watson Automated Speech
6 Recognition engine first as a member of the AT&T Bell Labs Speech Recognition Research team
7 and then as a member of AT&T Research's Speech Processing Software & Technology Research
8 department.

9 5. I was manager of Nortel's Speech Recognition Research Group from 1990 to
10 1994, where I led the creation of the technology required for the Automated Directory Assistance
11 System (ADAS+), a Nortel product that provides full or partial automation of DA calls.

12 6. I have over 15 years experience in Interactive Voice Technologies for
13 Telecommunications, and have been granted four speech technology patents.

14 **B. Nuance's Speech Recognition System Version 6**

15 7. Nuance was spun out from Stanford Research Institute ("SRI") and was
16 incorporated in 1994. Nuance has been selling speech recognition software since that time.
17 Nuance's software allows a customer to implement a speech-enabled interactive voice response
18 system that would help a user retrieve information—such as a current balance—or perform an
19 action—such as transferring money – all through voice commands. For example, the Nuance
20 Speech Recognition System Version 6 Developer's Manual explains that:

21 A speech recognition application is any piece of software that uses speech
22 recognition to help a user retrieve information or perform an action. One example
23 of a speech recognition application is a system that a bank customer can call to
24 ask about a current balance. Such a system politely leads the customer through an
interaction, asking questions, understanding the caller's responses, and providing
information. The system has many components; the component that listens and
understands is the speech recognition engine.

25 The Nuance speech recognition engine is both powerful and flexible. It provides
26 speaker-independent speech recognition capability, can recognize from lists of
15,000 or more phrases, and can extract meaning from naturally spoken
27 sentences. As a speech recognition application developer, you can configure the
Nuance speech recognition engine to meet the needs of your application. This
28 configuration is called a recognition package.

1 A recognition package configures the recognition engine to perform a set of tasks
2 that are appropriate to the application. For instance, if an application asks "Is this
3 correct?", the recognition engine must be prepared to recognize "yes", "yeah",
4 "yes it is", "no", "no it's not", "nope", and so forth. This set of phrases is called a
recognition grammar, and a recognition package is a set of such grammars,
combined with a set of configuration parameters that allow you to maximize
accuracy and minimize CPU use.

5 Nuance Developer's Manual, Version 6, p. ix.

6 8. Nuance Speech Recognition System ("NSRS") version 6 has been on sale since
7 1997. NSRS version 6 was capable of speaker-independent, continuous speech recognition.
8 Continuous speech recognition refers to the fact that recognizer does not require artificial pauses
9 between words to recognize speech. NSRS version 6 is accurately described in the Nuance
10 Developer's Manual version 6 (Bates No. WF625) that is attached as Exhibit A hereto.

11 **1. NSRS Architecture**

12 **a. Recognition Client**

13 9. NSRS version 6 used a recognition client and recognition server to recognize
14 speech. The recognition client was capable of coordinating the speech recognition process and
15 performing the echo cancellation, speech detection, capture, and endpointing functions. This is
16 described in Nuance Developer's Manual version 6 on p. 78: "The RecClient . . . provides the
17 application with the following functionality: . . . Energy-based endpointing to determine the
18 beginning and end of the talker's speech." Endpointing uses certain thresholds—typically the
19 energy level in the audio signal representing the callers speech at a given time—to identify when
20 a caller is speaking (and when a caller is not). Endpointing is essentially a more robust form of
21 voice activity detection. By using endpointing, the recognition client could send only speech
22 data (and not silence) to the recognition server. Echo cancellation refers to software that
23 eliminates echoes in the incoming signal.

24 10. Endpointing was configurable in NSRS version 6. A developer was able to either
25 run endpointing on an input utterance, or alternatively, disable endpointing and simply specify
26 the start and end time of the utterance to be recognized. This is described in the Nuance
27 Developer's Manual version 6 on p. 131: "The RecClient API also allows the developer to
28 specify the start and end of the utterance to be recognized, bypassing the spectral energy

1 endpointing used by RCRecognize (). This is also shown by Nuance Developer's Manual
2 version 6 on p. 329: "When endpointing is not performed, all samples received by the client
3 process are sent to the recognition server. Because users often pause after they are prompted to
4 speak, some silence samples are unnecessarily sent to the recognition server. This increases the
5 load put on the recognition server and lengthens the delay before the recognition server can
6 respond." Further, there are a set of parameters that a developer could adjust to control the
7 behavior of the endpointer. This is described on page 330 of the Nuance Developer's Manual.

8 11. In NSRS version 6, the recognition client was capable of being run on a different
9 machine from the machine running the recognition server to complete speech recognition. This
10 is described in Nuance Developer's Manual version 6 on p. 85: "Distributed Architecture.
11 Recognition, which can be CPU-intensive, can be run on a different machine from the machine
12 running the application and audio interface." This is not required, and a developer could decide
13 to run the recognition client and the recognition server on the same machine.

14 12. In NSRS version 6, the recognition client was capable of being run on, or in
15 conjunction with, a telephony board—which I understand that Phoenix has called a "speech
16 board." This is shown in the Nuance Developer's Manual version 6 on p. 135: "The Nuance
17 System RecClient supports several telephone interfaces, including Dialogic." For example, the
18 Dialogic Antares board was capable of performing echo-cancellation and endpointing. This is
19 described in Nuance Developer's Manual version 6 on p. 261: "The assumption is that the
20 telephony card itself is being controlled by some third-party software, and the Antares is being
21 used by Nuance software to perform echo cancellation and endpointing." Further, the Dialogic
22 Antares boards were capable of being configured to handle more or fewer signal processing
23 functions. For example, a print out from a Dialogic website dated October 1999 that was
24 retrieved using archive.org (Bates No. WF587) states:

25 "Rather than offering a specific high level product, DSPSE [DSP Software
26 Engineering, Inc.] offers the means to create custom products based on DSP
27 technology with the Antares platform. DSPSE has 21 algorithm products that are
28 compatible with the Antares platform and which can be used to quickly build high
performance applications that fit specific customer requirements. The Systems
Group at DSPSE has used these algorithm products to build "private label"
solutions on the Antares platform. The combination of Antares specific

1 experience and expertise in the development of DSP algorithms and applications
2 enables DSPSE to quickly create the exact DSP based solution your application
3 requires.”

4 This statement accurately reflects my understanding of a developer’s ability to adjust the
5 functions performed by the Dialogic Antares board.

6 13. Further, such telephony boards support audio input and output for the NSRS.
7 Accordingly, these boards were responsible for receiving voice signals coming from connected
8 telephone/network lines, digitizing those signals, and then passing along the live audio data to
9 the recognition client. This is described in Nuance Developer’s Manual version 6 on p. 80: “An
10 audio provider, a set of C interfaces that handle the reception and transmission of live audio data
11 to and from the current hardware. Audio providers are described in Chapter 15, ‘Nuance Audio
12 and Telephony Providers.’”

13 14. As mentioned above, a developer could run the recognition client and the
14 recognition server on different machines in NSRS version 6. Accordingly, the recognition client
15 included the ability to format and transmit the speech data after endpointing to the Nuance
16 recognition server. This is described on the Nuance Communications website dated April 10,
17 1998 (Bates No. WF595): The resulting “speech values are then sent over a network connection
18 to the RecServer which handles the speech recognition and understanding.”

19 15. Further, the recognition client sent speech data to the recognition server during
20 the period when the caller was talking. In other words, the recognition client did not wait to send
21 speech data to the recognition server until the entire input was captured. This is described in
22 Nuance Developer’s Manual version 6 on p. 329: “Samples begin flowing to the recserver as
23 soon as the onset of speech is detected, and the flow continues until the end of speech is found.
24 In this way, the recserver can actually begin to process the utterance while the person is still
25 speaking.”

26 **b. Recognition Server**

27 16. In NSRS version 6, the recognition server received the data transmitted from the
28 recognition client, and thus the NSRS contained a routine that received speech data from the
recognition client. The recognition server was also called the RecServer, and it completed

1 speech recognition and understanding.

2 17. The RecServer used Hidden Markov Models (HMMs) to recognize speech. This
3 is described in Nuance Developer's Manual version 6 on p. 325: "Naturally, the Nuance System
4 uses state-of-the-art HMMs at the core of its recognition engine. HMMs function primarily as
5 acoustic models that provide a mapping from the sampled speech signal to a sequence of
6 phonetic units." In the NSRS version 6 system, the recognizer finds the best match between the
7 input speech and a path through the HMMs. This path can be used to determine the recognizer's
8 best estimate of the words spoken by the caller. This is described in Nuance Developer's
9 Manual version 6 on p. 349: "The Nuance Speech Recognition System represents each word in
10 the recognition vocabulary as a sequence of phonetic symbols. These symbols provide the
11 correspondence between a word in the dictionary and its pronunciation." This is described in
12 Nuance Developer's Manual version 6 on p. 15: "To recognize speech, the Nuance System
13 compares it with a graph of acoustic speech models. In a process called the recognition search,
14 the recognition engine searches this graph of possibilities for the sequence of models that best
15 correspond to the speech. These models are found and interpreted as strings of words, and
16 returned as the recognition result." After processing, the recognition server was capable of
17 returning a result that represented the recognized speech, and was capable of doing so in real
18 time.

19 18. The NSRS version 6 also was capable of returning partial results. This is
20 described In Nuance Developer's Manual version 6 on p. 239: "Set this Boolean parameter to
21 TRUE if you want the recognition engine to generate partial (intermediate) recognition results
22 periodically. Each result will generate a NUANCE-EVENT-PARTIAL-RESULT." This
23 function would permit (among other things) the recognizer to provide feedback in real time so
24 that, for example, the system could stop playback of a prompt.

25 19. Further, in NSRS version 6, a developer was able to configure the degree of
26 pruning. The degree of pruning refers to the intensity of the recognition search. The developer
27 in this way controls the speed/accuracy trade-off by varying the intensity of the search performed
28 by the recognizer to find the spoken word. Essentially, the degree of pruning adjusts how many

1 options the speech recognition engine examines to find its answer as to what word was spoken
 2 by a caller. This is described in Nuance Developer's Manual version 6 on p. 15-16: "rec.Pruning
 3 This parameter controls the trade-off between speed and accuracy by constraining the search
 4 space. By limiting the number of different search possibilities carried along, recognition speed
 5 can be increased."

6 20. Further, in NSRS version 6, the recognition server was able to understand
 7 different forms of a word such as "Give me" and "Gimme" and phrases such as "give me the
 8 cheapest flight." This is described in Nuance Developer's Manual version 6 on p. 22-23: "For
 9 example, users will often say "Give me the cheapest flight" as "Gimme the cheapest flight." You
 10 can improve accuracy by explicitly including these compound words with additional paths in
 11 your grammar, for example: [(give me) give-me] the cheapest flight) and augmenting your
 12 dictionary, for example: give-me g ih m iy." The recognition server was able to recognize
 13 speech despite the presence of dysfluencies such as "ahhh" and "ummmm" or other words
 14 extraneous to the caller's request. Moreover, in the recognition server, a designer was able to
 15 assign different weights to words when including them in the grammar; the grammar specified
 16 the strings of words that the system could recognize.

17 c. Natural Language Component

18 21. NSRS version 6 also had an optional natural language component in a speech
 19 recognition system. If a particular system implemented that component, the natural language
 20 component took a sentence (typically a recognized utterance) as input and returned an
 21 interpretation—a representation of the meaning of the sentence. This is described in Nuance
 22 Developer's Manual version 6 on p. 39: "The Nuance System allows developers to quickly and
 23 easily create powerful natural language understanding systems tailored to their needs. A natural
 24 language understanding system takes a sentence as input and returns as output an interpretation,
 25 which is a representation of the meaning of the sentence. . . ."

26 22. If implemented, the natural language component attempted to match spoken
 27 words to a fixed set of slots. This meant that a designer could set up slots representing what she
 28 believed to be the important parts of a spoken sentence. In a banking application, for example,

these slots could include a slot to represent the command (*i.e.*, transfer, transferring, send, sending, etc.), a slot to represent the amount (*i.e.*, five hundred), a slot to represent the source account (*i.e.*, checking), and a slot to represent the destination account (*i.e.*, savings). The natural language component would then determine whether the recognized words corresponded to any slots to figure out what the caller meant, for example by saying “please send five hundred dollars to savings from checking” or “I would like to go about transferring from saving to checking five hundred dollars.” This is described in Nuance Developer’s Manual version 6 on p. 39: “Natural language understanding applications built with the Nuance System produce interpretations consisting of slots and values for these slots. For any given application, the developer specifies a fixed set of slots, which correspond to types of information commonly supplied in queries in the application’s domain. For example, in an automated banking application, slots might include command-type, amount, source-account, and destination account.” Accordingly, the recognition server was able to understand different forms of words to relate to the same specified slot to fill the command-type slot—*e.g.*, either transfer and transferring. Likewise, the recognition server was able to understand phrases such as “transfer five hundred dollars from savings to checking” This is described in Nuance Developer’s Manual version 6 on p. 39: “The interpretation for the command ‘Transfer five hundred dollars from savings to checking’ would then probably fill the command-type slot with transfer, the source-account slot with savings, the destination-account slot with checking, and the amount slot with 500.” Moreover, in selecting a fixed set of slots, the designer was able to assign different weights to words in the recognition server—*e.g.*, the ones that corresponded to the slots.

23. Further, if implemented, the natural language component also was able to consider words outside of the defined slots and still determine an appropriate response to a customer’s query. Correspondingly, the NSRS version 6 was capable of understanding spoken queries that include dysfluencies and still determine the response to the query.

d. Context Parameters

24. NSRS version 6 included the ability to consider the context of a caller’s utterance in recognizing the words in that utterance. For example, in a banking application, if a caller said

1 “transfer that amount today”, the recognition server was able to understand that “today” referred
 2 to a particular date based on the context of the utterance. This is discussed in Nuance Developer
 3 Manual version 6 on p. 119 (“Contextualization refers to the process of updating something to
 4 reflect the context of utterance. The function ContextualizeDate () updates a date value to reflect
 5 the time at which it was uttered.”). Further, the NSRS allowed where a caller was in the
 6 dialogue to play a role in recognizing the caller’s speech—*e.g.*, the recognition server knew if the
 7 caller had just told the system that she wanted to transfer money and thus interpreted the caller’s
 8 next speech in light of that knowledge. This is accomplished using the AppSetGrammar as
 9 described on p. 112. Further, NSRS version 6 included the ability to dynamically load a
 10 particular grammar or a set of word pronunciations (a dictionary) to be used in understanding a
 11 caller’s query depending on context. This is described in Nuance Developer Manual version 6
 12 on p. 279 (“The Nuance System allows the dynamic creation and modification of recognition
 13 grammars from within a running application. This capability enables many new types of speech
 14 recognition applications, such as: A speed dialer, in which the user’s own grammar of names
 15 (such as “mom”, “the office”, and “robert jones”) is dynamically loaded when that user calls; a
 16 database front end, in which the application looks at the set of currently selected items, and
 17 dynamically builds a recognition grammar that allows the user to select from among those
 18 items.”) and on page 289(“WGISetPronunciation”).

19 **e. Database Interface**

20 25. After recognition, NSRS version 6 was capable of generating a database query to
 21 retrieve answers in response to a caller’s query. This is described in Nuance Developer’s
 22 Manual version 6, p.120 (“Two database systems are currently supported: Informix and Mini
 23 SQL. These are both relational database systems that process SQL queries. . . . If you pass the
 24 argument -db database-name into AppNew () , the system will automatically connect to the
 25 given database (and disconnect when the . . .object is deleted).”) and on p. 296 (“This function
 26 generates the query from the active interpretation of the given NLResult object. The query
 27 produced is placed in the query buffer, provided that it fits.”).

2. NSRS Operation

26. NSRS version 6 was implemented in a number of systems that retrieved answers in response to a caller's query. For example, Nuance implemented NSRS version 6 to provide speech-enabled demonstrations where people could call a phone number and experience the NSRS's ability to recognize their question and respond appropriately. This is shown, for example, on the Nuance website that described the Nuance stock quote demonstration which "shows Nuance's ability to provide highly accurate quotes on over 13,000 stocks, mutual funds, and market indicators. To reach the demo, call 650-847-7423. Simply say the name of the company, fund, or market indicator." (WF603).

27. NSRS version 6 also was implemented in a demonstration banking system that covered topics such as "account balance" among other topics such as "transfer money" and "pay bills." This is shown, for example, on the Nuance website on p. 5-6 (WF599-600) and in the Nuance Developer's Manual version 6 on p. 317 which provides "a sample banking application built with the Dialog Builder [that] lets you check your balance, transfer money, pay bills, and add new payees. The "add payee" feature illustrates the enrollment and dynamic grammar capabilities of the Nuance software." The Nuance demonstration systems knew how to give an appropriate answer when it figured out what the query/topic is all about. A caller could always to conduct a second session with the demonstration to get additional answers on the same topic (for example, account history) by calling the system again.

28. Further, a prior art NSRS also was implemented in the Charles Schwab VoiceBroker system that responded to speech-based queries from customers concerning certain banking and investment topics. This is described in a 1996 VoiceBroker Press Release where Nuance announced the VoiceBroker system "to provide stock, mutual fund and market indicator information to retail customers." (WF593). Such a NSRS was capable of being implemented by a company, such as Charles Schwab, that also maintained a website that included the ability to select items (using mouse clicks in a browser) that also could be selected through a separate speech interface. I confirmed that Charles Schwab did in fact have such a website by searching archive.org.

1 29. All of the above-mentioned NSRS implementations were capable of returning
2 responses in audible spoken responses. Indeed, NSRS version 6 was capable of operating in
3 conjunction with Entropic's TrueTalk text-to-speech system that would play the spoken
4 responses. This is described in Nuance Developer's Manual version 6 on p. 121: "The only
5 text-to-speech system that Nuance has an interface to is Entropic's TrueTalk." Moreover, NSRS
6 version 6 also was capable of playing audio files that included responses to a caller. This is
7 discussed in Nuance Developer's Manual version 6 on p. 319: "The Nuance System includes a
8 waveform editor, Xwavedit, which makes it easy to record, play, view, crop, cut, and paste
9 waveforms. The Xwavedit program is ideal for recording prompts and removing trailing silence
10 from them."

11 30. Further, NSRS version 6 was capable of being used in a system that provided a
12 prompt regarding possible user queries, confirmed the substance of the query, and provided an
13 answer (or completed the requested action). The Nuance Developer's Manual version 6 on p. 99
14 describes such a system:

15 "To build an application with the Dialog Builder, you create a set of dialog states.
16 A dialog state provides code that handles a single interchange between the user
17 and the computer. For example, a banking application might have a state in which
18 the application asks the user "How much do you want to transfer?" and the user
19 responds with a dollar amount that the system would recognize. After a successful
interchange, an application typically moves into a new dialog state-for example,
the banking application might move into a confirmation state after the transfer
amount state. You can envision the collection of interconnected dialog states in an
application as a network.");

20 This is also discussed on p. 103-104 where the Manual teaches how to configure the NSRS to
21 play a prompt ("file that contains a recording of the prompt 'How much do you want to pay?")
22 and to confirm the request ("The system prompts something like 'You specified fifty dollars").
23 The system then would complete the requested action.

24 31. Because the designer could record and play any audio file in NSRS version 6, a
25 designer was able to record prompts. A developer could record a female voice for the prompt, if
26 the developer believed that was his or her customer's preference. This is described in the
27 Nuance Developer's Manual version 6 on p. 319 ("The Nuance System includes a waveform
28 editor, Xwavedit, which makes it easy to record, play, view, crop, cut, and paste waveforms. The

1 Xwavedit program is ideal for recording prompts and removing trailing silence from them.”) and
2 on p. 321 (“Nuance provides several programs you can use to normalize .wav files [including the
3 ability to] determine the average energy level of speech in a set of .wav files [and] . . . perform
4 operations such as clipping and filtering on .wav files.”).

5 32. NSRS version 6 also included the capability to monitor the system performance.
6 This is described in the Nuance Developer’s Manual version 6 on p. 24: “One way to check the
7 speech signal is to use Xwavedit to examine recordings made by your application. You should
8 listen and look for endpointing errors, signal distortions (e.g., clipping), very low amplitude
9 signals, and recordings that just sound bad.”


10 33. Finally, prior to November 12, 1999, NSRS was capable of being operated in
11 conjunction with Nuance Voyager such that people were able to place phone calls and navigate
12 between those calls and voice-enabled Web sites using spoken hyperlinks. This is described on
13 Nuance’s website dated November 1999 that was retrieved using archive.org which explains
14 that: Nuance Voyager “is a voice browser that significantly enhances the capabilities of the
15 telephone through a voice interface that parallels that of a desktop Web browser. With Voyager,
16 people can place phone calls and navigate between those calls and voice-enabled Web sites using
17 spoken hyperlinks, or “voicelinks.” Voyager is designed to allow phone users to tap into the
18 wealth of information available on the Internet, all by using their voice and without hanging up
19 the phone.” (WF609).

20 34. Indeed, in November 1998, Nuance entered an alliance that enabled users to
21 conduct business over the Web, such as booking a plane reservation, and then access the same
22 information over an ordinary telephone, such as changing that reservation while you are on the
23 road. This alliance is reported below:

24 “Nuance Communications recently teamed with Motorola Inc. and Visa
25 International to supplement Web-based commerce with speech-recognition
26 capabilities. The technology will allow Web-based businesses to add integrated
27 speech-recognition access to their services over ordinary phone lines. Three
28 technologies were blended—Nuance speech objects, Motorola’s Voice Markup
Language (VoxML) and a Java application programmer interface—to allow users
equal access to businesses from either the Web or the telephone. This alliance
enables users to conduct business over the Web, such as booking a plane
reservation, and then access the same information over an ordinary telephone,
such as changing that reservation while you are on the road.”

1 (WF611).

2 I declare under penalty of perjury that the above statements are true and correct to the
3 best of my knowledge and that I executed this declaration on September 5, 2008 in Redwood
4 City, California.

5 
6 R. Douglas Sharp

SHARP DECLARATION

EXHIBIT A

(Part 1 of 2)

*nuance Speech
Recognition System*

developer's manual





NUANCE SPEECH RECOGNITION SYSTEM

Version 6

Developer's Manual

.....

Nuance Speech Recognition System, Version 6
Developer's Manual

Copyright © 1997 Nuance Communications, Menlo Park, California. All rights reserved
Reprinted 1998

Information in this document is subject to change without notice and does not represent a commitment on the part of Nuance Communications. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Nuance Communications.

Nuance, RecClient, and RecServer are trademarks of Nuance Communications.

BSD is a product of UNIX Systems Laboratories, Inc. and the University of California.

Computerfone is a trademark of Suncoast Systems.

Dialogic, Antares, and GlobalCall are trademarks of Dialogic Corporation.

IBM and DirectTalk are registered trademarks of International Business Machines Corporation.

Informix is a registered trademark of Informix Software, Inc.

Intel is a registered trademark of Intel Corporation.

Periphonics and PeriProducer are trademarks or registered trademarks of Periphonics Corporation.

Purify is a registered trademark of Pure Atria

SCO is a trademark of The Santa Cruz Operation, Inc.

Silicon Graphics is a registered trademarks of Silicon Graphics, Inc.

Sound Blaster is a registered trademark of Creative Technology Ltd.

SPARC is a registered trademark of SPARC International, Inc.

Sun and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc.

TrueTalk is a trademark of Entropic Research Laboratory, Inc.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

Voicetek and VTK are registered trademarks of Voicetek Corporation.

Windows and Windows NT are trademarks or registered trademarks of Microsoft Corporation.

Any other product names mentioned herein are the trademarks or registered trademarks of their respective owners.

Nuance Communications contributors: Eric Chang, Mike Hochberg, Eric Jackson, Mark Klenk, Debbie Knoles, Tom Kuhn, Chris Leggetter, Peter Monaco, Hy Murveit, Keith Rolle, and Ben Shahshahani.

Printed in the United States of America

USING THIS MANUAL

A *speech recognition application* is any piece of software that uses speech recognition to help a user retrieve information or perform an action. One example of a speech recognition application is a system that a bank customer can call to ask about a current balance. Such a system politely leads the customer through an interaction, asking questions, understanding the caller's responses, and providing information. The system has many components; the component that listens and understands is the *speech recognition engine*.

The Nuance speech recognition engine is both powerful and flexible. It provides speaker-independent speech recognition capability, can recognize from lists of 15,000 or more phrases, and can extract meaning from naturally spoken sentences. As a speech recognition application developer, you can configure the Nuance speech recognition engine to meet the needs of your application. This configuration is called a *recognition package*.

A recognition package configures the recognition engine to perform a set of tasks that are appropriate to the application. For instance, if an application asks "Is this correct?", the recognition engine must be prepared to recognize "yes", "yeah", "yes it is", "no", "no it's not", "nope", and so forth. This set of phrases is called a *recognition grammar*, and a recognition package is a set of such grammars, combined with a set of configuration parameters that allow you to maximize accuracy and minimize CPU use.

2

TUNING RECOGNITION PERFORMANCE

To recognize speech, the Nuance System compares it with a graph of acoustic speech models. In a process called the recognition *search*, the recognition engine searches this graph of possibilities for the sequence of models that best correspond to the speech. These models are found and interpreted as strings of words, and returned as the recognition result.

The acoustic models are those specified to *nuance-compile* when a package is created (see Chapter 1). The graph is defined by the word grammars that the application developer writes and passes to the *nuance-compile* program.

The accuracy and speed of the Nuance System is determined by several factors:

- *The input speech: noise level, distortions, speech clarity, and so forth.* The more clean the speech, the faster and more accurate the recognition.
- *The complexity of the acoustic models.* More complex models are more accurate but often result in slower recognition times.
- *The complexity of the grammar.* A large or complex grammar usually results in a slower recognition system. The recognition accuracy may be worse, because of the larger number of possible word sequences. However, such a grammar may be more accurate if the grammar better reflects actual input speech.

- *The confusability of the grammar.* Typically, words that sound similar are difficult to distinguish, and grammars depending on such distinctions result in higher error rates.
- *The amount of search performed.* The recognition system can be configured to perform less search. This will speed up the system but might cause search errors, if good theories are missed.

The performance of the Nuance System can be optimized by varying system configurations. In general, the default values of Nuance recognition parameters are acceptable for most applications, but performance for a particular application can be improved through experimentation. You are encouraged to collect speech recordings of application usage and to measure the accuracy and speed of the recognizer under various configurations by using the *batchrec* program. Chapter 3 explains how to use *batchrec* and how to interpret its output. This chapter introduces the parameters of the recognizer and provides some hints on tuning these parameters.

CREATING EFFECTIVE GRAMMARS

The art of speech recognition application design lies in creating an intuitive user interface that guides the user through a constrained, yet productive, interaction. The user speaks naturally, yet, because of well-designed prompts and grammars, the user's input fits within the expectations of the recognition grammar, and the system achieves a high recognition accuracy rate.

When designing grammars for a particular application, keep the following in mind:

1. *Coordinate grammars and prompts.* Remember that the prompt and the grammar must be considered together when you design either one. The prompt given to a user strongly affects the way the user speaks, even in subtle ways. For instance, if you are designing the grammar for a yes/no question, and your prompt is:

"Is ... true?"

then, among other possibilities, the grammar should allow the response:

"Yes it is."

If the prompt were:

"Do you want to?"

then the grammar should certainly allow:

"Yes I do."

2. *Consider all the likely ways a user can answer the prompt.* Reducing grammar size improves the speed and in-grammar recognition accuracy of the system. Unfortunately, it may also hurt accuracy if users tend to stray from the prescribed grammar. Experience shows that user input that does not match system grammars is often the largest source of error in a speech recognition application. Trading off advantages of very strict grammars (e.g., just "yes" or "no") versus flexible grammars is an important part of application design. Issues such as in-grammar accuracy, user-speech coverage, out-of-grammar rejection, and processing speed can all be measured and should all enter into your design equation.
3. *Do not try to cover too much.* Your grammar should not attempt to cover all the spoken English language. Focus on creating applications and grammars that constrain the input utterances to just the domain at hand. Focus on designing a good user interface that helps the user provide reasonable speech input for reliable recognition performance.
4. *Consider reducing the scope of the question.* You can trade off the amount of information given at a certain point. For example, if you are developing an application that recognizes the name of a city in the United States, you may design the application to first ask for the state name. The extra burden of an additional question to the user is traded off with a potential speedup and accuracy enhancement. The proper decision is application-dependent and is an important one for you to consider.

CHOICE OF ACOUSTIC MODELS

The Nuance System's sets of acoustic-phonetic hidden Markov models are described below. The algorithm development team at Nuance Communications frequently releases new acoustic models, so this manual cannot describe the specific models included in this release. The usage line

printed by *nuance-compile* prints the currently installed model sets. Models can be grouped according to their underlying technologies:

- *Genone* models—whose names start with “gen”—are the Nuance System’s most accurate continuous-density HMMs.
- *Phonetically tied mixture* (PTM) models are also continuous-density HMMs, but PTM models use fewer Gaussians than Genone models, and are generally faster but slightly less accurate.
- *Vector-quantized* (VQ) models use discrete-density probability functions. VQ models can be the fastest models that Nuance Communications ships, but they tend to make more errors than either PTM or Genone models.

Generally, each release of the Nuance System includes one set of each type of acoustic model. You should probably begin development by using PTM models for large vocabularies and Genone models for smaller vocabularies. However, each set of models will perform differently for a given recognition task. To optimize the performance for your application, Nuance Communications recommends that, as development progresses, you accumulate a large set of utterances collected during actual use of your application. These utterances can then be run in batch mode using the *batchrec* program to evaluate the speed and accuracy of each model set so that you can make an informed decision.

Nuance Communications will also release models targeted for different languages or dialects of a language.

TUNING SEARCH PARAMETERS

Three parameters affect the way the Nuance System performs the search for the optimum word sequence in the grammar. Fine tuning these parameters, as explained here, can significantly improve the speed and accuracy trade-off for your application.

`rec.Pruning`

This parameter controls the trade-off between speed and accuracy by constraining the search space. By limiting the number of different search possibilities carried along, recognition speed can be increased. However, once a possibility is eliminated from consideration, it can never be brought

back. Therefore, tightening the search beam can increase the recognition error rate. Each model set comes with a preset default pruning value. All the current Nuance System model sets should use pruning values in the range of 600 to 1000 for systems without grammar probabilities. A larger pruning value means that more hypotheses will be considered.

`rec.PPR`

This Boolean parameter controls the trade-off between speed and accuracy by looking ahead to prune less likely hypotheses from further consideration during the search. In doing so, the recognizer performs extra computations using approximate models. If the grammar is large, the pruning of unlikely hypotheses compensates for the extra computations needed and, hence, the net effect is a faster response time with slightly worse accuracy. On the other hand, if the search space is small, such as in a digit recognition or a small-vocabulary command recognition task, the extra computations might cause longer response time. Therefore, the choice of TRUE or FALSE for these parameters depends on the grammar and application at hand. It is recommended that this parameter always be set to TRUE for large-vocabulary tasks, and FALSE for small-vocabulary tasks.

`rec.GrammarWeight`

This parameter controls the *grammar processing weight*—the relative weighting of acoustic and linguistic scores during recognition. This is meaningful only when probabilities in your grammar are used. (See Chapter 1 for more detail about grammar probabilities.) The relative weight of the grammar probabilities versus the acoustic scores is controlled by `rec.GrammarWeight`. A large value for this parameter emphasizes the linguistic constraints set by the probabilities, and a small value emphasizes the acoustic scores. The `rec.GrammarWeight` interacts directly with `rec.Pruning`. Therefore, if you decide to use probabilities in your grammar, carefully tune `rec.GrammarWeight` and `rec.Pruning` jointly by performing *batchrec* experiments with a variety of settings.

- **Note:** Grammar probabilities are most often used for those large-vocabulary applications where probabilities can be estimated accurately. Most developers never use probabilities and, therefore, never modify `rec.GrammarWeight`.

RECOMMENDATIONS FOR PARAMETER SETTINGS

Optimum parameter values depend on the application, the user interface requirements, and the hardware platform. You should collect example utterances and perform recognition experiments by using the *batchrec* program with various parameter settings, and then choose the optimum set of values. In doing so, the following tips are useful.

Recommendations for Small-Vocabulary Applications

Applications with vocabulary sizes smaller than 50 are usually considered *small vocabulary*. For small-vocabulary applications the extra computations needed for performing pruning through the use of `rec.PPR` parameters can slow down the system. Therefore, it is usually better to set this parameter to `FALSE`. The `rec.Pruning` parameter should be tuned carefully by experimentation. In addition, the choice between various acoustic models (Genone, PTM, VQ) depends on the acoustic confusability of the words in the vocabulary and must be made through experimentation. For recognizing digits and numbers, Genone models are recommended.

Recommendations for Large-Vocabulary Applications

Applications with vocabulary sizes larger than 1000 are usually considered *large vocabulary*. Large-vocabulary applications can benefit from phonetic pruning by setting the `rec.PPR` parameter to `TRUE`. If probabilities for various words in the vocabulary (or subgrammars) can be accurately estimated, the speed and accuracy trade-off may be improved by jointly tuning `rec.GrammarWeight` and `rec.Pruning`. In addition, large grammars, especially the ones that contain subgrammars frequently used by other grammars, create packages that consume a large amount of memory. Applying the option `-dont_flatten` to *nuance-compile* can cause a significant reduction in the memory footprint of these packages (see page 12 for more details). The `-dont_flatten` option will slow recognition slightly for some applications.

ENDPOINTING

The Nuance System removes leading and trailing silence from spoken utterances before sending them to the recognition engine. This process is called *endpointing*, and serves to allocate recognition resources only when they are needed. The parameters that affect the behavior of the endpointer are described more extensively in the section "Endpointing" on page 329. The optimum values for these parameters are application- or grammar-dependent. In particular, the parameter `ep.EndSeconds` controls the minimum amount of silence, after a spoken phrase is uttered, that the system observes before it will decide that the talker has finished speaking. A large value for this parameter (for example, 1 to 2 seconds) makes the system less likely to cut off the talker. It is suitable for situations when the talker might pause mid-sentence—for example, when entering long account numbers. Smaller values (for example, 0.75 second or less) are appropriate for simple questions. Smaller values make the interaction more snappy, and may deter the system from combining extraneous speech into the utterance to be recognized, but increase the possibility of cutting off the user in mid-sentence.

The parameter `ep.StartSeconds` controls the minimum amount of high-energy speech needed to detect speech. It defaults to 0.15 second. Nuance Communications has found this to be the ideal value in most situations.

The parameters `ep.AdditionalStartSilence` and `ep.AdditionalEndSilence` control the amount of padding the system uses to minimize the effects of endpointer errors. That is, the system recognizes the waveform starting 0.3 second (the default for these values) before the start of speech is detected, and ending 0.3 second after the end of speech is detected. To be clear about this, once the system sees `ep.StartSeconds` seconds of high energy, it labels the start of speech at the start of that high-energy region and goes back another `ep.AdditionalStartSilence` seconds. Then, when the system detects `ep.EndSeconds` of silence, it labels the end of speech at the start of that region and tacks on `ep.AdditionalEndSilence` of silence.

You should listen to recordings made by the system if you suspect endpoint errors. The recordings should start `ep.AdditionalStartSilence` before the start of speech and end `ep.AdditionalEndSilence` after the end. Endpointing errors are easy to hear. If you notice that on many occasions the end of the sentence is missing, consider increasing `ep.EndSeconds`. If the

start of speech is missing, this typically indicates that the user began speaking before the system was listening. This can occur in applications that do not allow *barge-in* (see "Barge-In" on page 331), particularly if the prompt has some silence at the end. You should make sure to trim all silence off the ends of prompts, so that the user does not prematurely think it is time to talk. If this endpoint error continues to occur, and you chose not to use barge-in, you should consider rewording your prompts to discourage talking ahead.

In some cases, the endpointer may have trouble detecting start of speech for words with unstressed or quiet first syllables. Increasing the value of `ep.AdditionalStartSilence` provides a work-around for this problem.

USE OF REJECTION MODELS

Callers interacting with a speech recognition system commonly speak utterances that are not covered by the recognition grammar. These out-of-grammar utterances include noises like coughs and door-slams, as well as spoken sentences that were not expected by the application developer who wrote the recognition grammar. In many cases, out-of-grammar speech is a large source of recognition error, as the recognizer may confuse the utterance with a valid, in-grammar possibility.

The default behavior of the Nuance System is to reject utterances not supported by the recognition grammar.

COMPOUND-WORD MODELING

To improve accuracy in sequences of short function words or in sequences that are often spoken quickly, you can add "compound-word" or "multi-word" pronunciations to your grammar and dictionary. For example, users will often say "Give me the cheapest flight" as "Gimme the cheapest flight." You can improve accuracy by explicitly including these compound words with additional paths in your grammar, for example:

```
((give me) give_me] the cheapest flight)
```

and augmenting your dictionary, for example:

```
give_me g ih m iy
```

Other examples of co-articulated “compound-word” pronunciations include:

what_are_the	w ah dx er dh ax
could_you	k uh d y ax
let_me	l eh m iy
what_is	w ah dx ih z

“Compound-word” modeling can also help in the recognition of sequences of very short words (each containing one or two phones). Using compound words in these situations allows the Nuance system to assign the most detailed context-dependent acoustic models possible. This usually yields better performance than using a sequence of individual words, where the many word boundaries result in the assignment of lower context-level (less detailed) models.

Examples of these so-called “cross-word” “compound-word” pronunciations include:

i_b_m	ay b iy eh m
a_t_and_t	ey t iy ax n t iy
e_mail	iy m ey l

In general, sequences of common and often under-articulated short words can be recognized more accurately when they are represented using these techniques.

The Nuance system also provides the capability of automatically constructing “compound-words” for all the words in a grammar. You enable this by specifying the `-do_crossword` option when running *nuance-compile*. This usually improves accuracy, but can also slow down recognition speed. This is only recommended for very small, mission-critical grammars (such as digits).

- **Note:** A disadvantage of using the “compound-word” approach alone is that pausing is not allowed between the individual words in these pronunciations. Therefore, for optimal coverage you should use “compound-words” in parallel with the original word sequences in your grammars.

TROUBLE SHOOTING

If you have problems with the accuracy or speed of the recognizer, the following tips might be helpful in identifying the problems.

High Error Rate

- Check the audio quality of the received speech waveforms. A bad audio channel can cause a significant degradation of the signal, which would in turn make the recognition difficult. One way to check the speech signal is to use *Xwavedit* to examine recordings made by your application. You should listen and look for endpointing errors, signal distortions (e.g., clipping), very low amplitude signals, and recordings that just sound bad.
- If you notice endpointing errors at the end of an utterance, try increasing `ep.EndSeconds` to see if the users are getting cut off. If you notice endpointing errors at the start of an utterance
 - Check to see if your prompt has silence at the end, and, if so, remove it using the *Xwavedit* program.
 - Reword and shorten your prompt to encourage your user to wait until the end of the prompt.
 - Add barge-in (enabling the speaker to interrupt prompts) to your application. See "Barge-In" on page 331 for more information.
- Make sure that the correct grammar is being used for recognition. When grammar files contain more than one top-level grammar, it is important to make sure that the correct grammar is being used during the recognition. In particular, when using the *batchrec* program (discussed further in Chapter 3), the top-level grammar to be used for the recognition is specified by inserting a line such as this into your *-testset* file:


```
*Grammar grammar-name
```
- Look for grammar coverage problems. Are the errors spoken sentences whose word sequence is not available in the grammar that was designed? This may be corrected by augmenting the grammar, or by improving the prompts being used, or by both techniques. You can use the *parse-tool*

program to help determine if sentences are being parsed correctly by your grammars (see the online documentation for usage information).

- Check to see if your grammar overgenerates. Overgeneration can cause errors if the correct sentences are being substituted for by nonsensible ones. You can see if this is happening simply by checking your recognition results, either from live applications or by using *batchrec*. You can check the grammar for overgeneration by examining the grammar, or by using the *random-generate* program (see the online documentation). This program will print random sentences that are allowed by the grammar. If you notice many nonsensible sentences, you can constrain your grammar.
- See if the errors you are observing are due to phonetically similar words in your grammar. If possible, avoid using acoustically similar words in similar grammar positions. In particular, homophones (words with identical phonetic pronunciations) are impossible for the recognizer to distinguish unless distinguished by the words around them in the grammar.
- Check to see if unexpected pronunciations are being allowed. For example, you might use the word “read” in your grammar, expecting only the present tense verb (pronounced “r iy d”) to be spoken. However, “read” also has the pronunciation “r eh d” in the main dictionary, which might easily be confused with another word (for example, “red”) in your grammar. Note that this can often happen with “a” being either the article “a” (generally “ah”) or the letter “a” (“ey”).
- Scale down your application. Sometimes it is beneficial to break the application into several steps in such a way that at each step a smaller grammar is used. For example, an application that intends to recognize the name of a company from the list of all the companies in all the stock exchanges can be scaled down so that the talker first indicates the name of the stock exchange. According to the recognized answer, the relevant subset of company names can be considered.
- Check to see if the words causing errors have incorrect or incomplete entries in the system’s or the package’s pronunciation dictionary. You may use the *pronounce* program (see the online documentation) to find pronunciations that are used in your recognition package. Incorrect pronunciations can degrade the performance of the speech recognizer.

-
- If errors occur in sequences of short function words, or there are word sequences that are often misrecognized when the user speaks quickly, consider adding “compound-word” pronunciations to the grammar and dictionary. “Compound-word” modeling is described on page 22.
 - If the recognizer is rejecting too many utterances, try setting the parameter `rec.ConfidenceRejectionThreshold` to a lower value. If the recognizer is accepting too many utterances that should be rejected, try setting this parameter to a higher value. The default value for this parameter is 45, but values between 35 and 55 may be reasonable depending on your grammar. Values outside this range tend to be extreme, with values higher than 60 rejecting too many utterances and values lower than 30 accepting too many utterances.
 - Check for excessive use of the @reject@ word. Remove it to see the impact it is having on accuracy. Try `rec.PruneRejects=TRUE`.
 - Increase the `rec.Pruning` parameter.
 - Set `rec.PPR=FALSE`.
 - Use more accurate acoustic models—that is, Genone models.

Slow Response Time

- Check for audio problems, which occur frequently in speech recognition systems. Noise may cause problems in the endpointer, or may slow down the search.
- Check to see if your grammar is unnecessarily large. You can do this by examining the grammar, or by using the *random-generate* program (see the online documentation). This will allow you to generate random sentences from the grammar. If you notice many nonsensical sentences, constrain your grammar.
- Check for endpointer errors. Slow response time can be due to the endpointer failing to quickly detect the end of speech.
- The perceived response time is affected by the parameter `ep.EndSeconds`. Using a smaller value for this parameter makes the

application snappier. Notice, though, that too small a value can cut off the talker.

- For medium- and large-vocabulary systems, try setting the parameter `rec.PPR=TRUE`.
- Check for excessive use of the `@reject@` word. Remove it to see the impact it is having on speed.
- Reduce the value of the `rec.Pruning` parameter.
- Use faster acoustic models (e.g., PTM models instead of Genones).
- Scale down your application. By breaking the application into smaller pieces, you can effectively reduce the complexity of the grammar being used at each point of the application. Less complex grammars run faster and more accurately.
- Make sure you have enough memory on your machine. Determine that your application is not paging when it is recognizing. If it is paging, you need either more memory or smaller grammars.

4

NATURAL LANGUAGE UNDERSTANDING: BASIC FEATURES

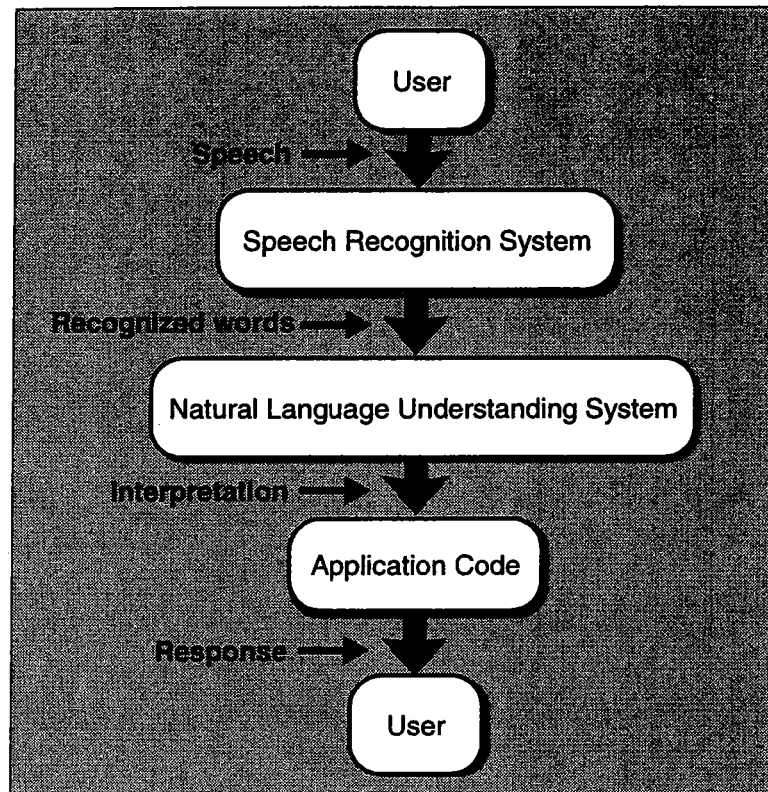
The Nuance System allows developers to quickly and easily create powerful natural language understanding systems tailored to their needs. A natural language understanding system takes a sentence as input and returns as output an interpretation, which is a representation of the meaning of the sentence.

INTERPRETATION

Natural language understanding applications built with the Nuance System produce interpretations consisting of *slots* and *values* for these slots. For any given application, the developer specifies a fixed set of slots, which correspond to types of information commonly supplied in queries in the application's domain. For example, in an automated banking application, slots might include *command-type*, *amount*, *source-account*, and *destination-account*. The interpretation for the command "Transfer five hundred dollars from savings to checking" would then probably fill the *command-type* slot with *transfer*, the *source-account* slot with *savings*, the *destination-account* slot with *checking*, and the *amount* slot with *500*. This interpretation could be depicted as follows:

```
{<command-type transfer> <source-account savings>
<destination-account checking> <amount 500>}
```

The natural language understanding system takes the recognized words as input and returns an interpretation.



A complete specification of a simple automated banking application is given on pages 47 and 48. This sample application illustrates the most important features of the natural language understanding system. The files that specify this application, *banking1.grammar* and *banking1.slot_definitions*, can also be found online in *\$NUANCE/sample-packages*. You may want to try modifying the application to verify your understanding of the material discussed here.

You might be wondering why (or whether) you need a natural language understanding system. The answer is that this system can make a developer's job much easier. Sentences of a natural language such as English are highly variable; the same information can be expressed in many different ways, and in different places within a sentence. In contrast, the interpretations produced by the natural language understanding system are both structured and simple. They are structured because information is divided into slot-value pairs; every value is associated with a certain slot, so you know what that value means. For example, account names in our automated banking application will be associated with either a *source-account* or *destination-account* slot, so you know what their role is in a user's query.

The Nuance System allows you to simply and concisely specify how interpretations are to be constructed from sentences, making it easy to create and modify applications. Furthermore, the specification of the application semantics is provided in the grammar file, the same location in which the recognition grammar is defined. This makes application development easier because it reduces the need for parallel changes in different places.

Like the speech recognition system, the natural language understanding system has two components, a compile-time component and a runtime component. The compile-time component takes the developer's specifications and compiles them. The runtime component reads the results of the compilation process, and then can repeatedly take recognized text as input and produce interpretations as output.

The following two sections describe the way in which the developer's specifications to the natural language system are expressed. Programs and API functions that provide access to the runtime component are described on pages 48 through 54.

ADDITIONS TO THE GRAMMAR FILE

The developer's specifications to the natural language system are primarily provided in the grammar file. Before reading this description, you should be familiar with the discussion of grammar files in Chapter 1. After reading this chapter, you may want to refer to the specification of the syntax of grammar files in Appendix C.

Commands and Attachment

The natural language understanding capability is specified largely by adding special *commands* to the grammar file. These commands are distinguished from the rest of the grammar by virtue of their being enclosed in curly braces ({ and }). A command attaches to the construction that precedes it, and is executed when the construction to which it is attached is matched against the input sentence.

For example, in the following grammar, the command in curly braces would attach to the word "from":

```
Phrase ( from {...} checking )
```

In the next grammar, the command would attach to the entire OR construction, [checking savings]:

```
Phrase ( from [ checking savings ] {...} )
```

Slot-Filling Commands

There are two types of commands. A “slot-filling” command specifies that a slot is to be filled with a particular value. The command has the following form:

<Slot Value>

For example, the following grammar specifies one way in which the *source-account* slot could be set:

```
Phrase (from [ ( ?my checking ?account)
                {<source-account checking>}
                ( ?my savings ?account)
                {<source-account savings>}
              ] )
```

Notice that although checking can be specified as a source account in four possible ways (“checking”, “checking account”, “my checking account”, and “my checking”), the *source-account* slot has only one corresponding value. This is a very simple example of how the system treats as equivalent different phrases that mean the same thing.

In the above example, the *source-account* slot was filled with a string value. It is also possible to fill a slot with an integer value:

```
Phrase ( transfer
        [ fifty    {<transfer-amt 50>}
          sixty    {<transfer-amt 60>}
        ]
        dollars
      )
```

If the natural language system can treat a value as an integer, it will do so. To force the value to be treated as a string, enclose it with double quotes. This is useful in digit grammars, where treating the value as an integer would lead to

the omission of leading zeros. For example, this would be the wrong way to write a digit grammar:

```
Digits [ ( zero zero zero){<digits 000>}
        ( zero zero one){<digits 001>}
        zero zero two){<digits 002>}
        etc.
]
```

This would be a better way:

```
Digits [ ( zero zero zero) {<digits "000">}
        ( zero zero one)  {<digits "001">}
        ( zero zero two)  {<digits "002">}
        etc.
]
```

Multiple commands may appear within curly braces. For example, the following is a legal grammar:

```
Phrase ( from savings to checking )
        {<source-account $savings>
         <destination-account checking>}
```

Return Commands and Variables

The second type of command, the *return* command, allows you to associate a value with a particular grammar, without filling any slots—for example:

```
Account [ ( ?my checking ?account) {return(checking)}
          ( ?my savings ?account) {return(savings)}
        ]
```

This could actually be written more concisely as follows:

```
Account ( ?my [ checking {return(checking)}
               savings {return(savings)} ]
        ?account )
```

This example also emphasizes the point that commands can appear anywhere in a grammar.

Return values do not appear in the interpretations produced by the natural language understanding system. Only filled slots appear in interpretations. However, return values can be placed into slots via the use of variables.

Variables are set to a return value by following a grammar name by a colon and then by the variable name. [Variable names can consist only of alphanumeric characters, the dash (-), the underscore (_), the single quote (') and the "at" sign (@).] Variables can then be referenced in a command by prefacing them with \$. Here is an example:

```
Phrase ( from Account:acct ) {<source-account $acct>}
```

This grammar says that the value of the variable *acct* should be set to the return value from the grammar *Account* and that the value of the *source-account* slot should be set to the value of *acct*. (Note that if we had omitted \$ before the variable *acct*, then the *source-account* slot would have been set to the string "*acct*"—which, of course, is not what we want.)

Return values are most useful when a certain type of value may appear in multiple slots. For example, we can now easily specify how the *destination-account* slot gets filled:

```
Phrase [ ( from Account:acct) {<source-account $acct>}
        ( to Account:acct ) {<destination-account $acct>}
      ]
```

THE SLOT DEFINITIONS FILE

In addition to the grammar file, you need a slot definitions file before you can use the natural language understanding capability. This file defines the slots for the application. The file must be entitled *application.slot_definitions* (where *application* is the name of the package being compiled) and must appear in the application directory. The slot definitions file is simply a list of all the slot names—for example:

```
source-account
destination-account
amount
command-type
```

Slot names must contain no whitespace. A semicolon indicates a comment; the semicolon and the rest of the line on which it appears are ignored by the compiler.

HOW THE DEVELOPER'S SPECIFICATIONS ARE APPLIED AT RUNTIME

The runtime component of the natural language system produces interpretations by matching grammars against the input sentence and executing the commands contained in those grammars. The commands executed are those attached to constructions that are part of the match. For example, consider the previous grammar example and suppose the input string is "to savings". The command `<destination-account $acct>` is executed because it is attached to the construction `(to Account:acct)`, which is matched against the input string. But the command `<source-account $acct>` is not executed because the construction to which it is attached, `(from Account:acct)`, is not part of the match.

The natural language system will produce an interpretation even if it cannot match the entire sentence against a grammar. When it can match the entire sentence against a grammar, it will do so, but if it cannot, it will return the interpretation that corresponds to the match of the maximum number of words possible against the sentence.

The natural language system provides information that indicates how well it was able to match the input sentence. If desired, the application can ignore those interpretations which do not correspond to perfect matches of the sentence.

Sometimes grammars are ambiguous—that is, the same sequence of words can produce more than one interpretation. This is the case with the following grammar:

```
.Command (call Name:nm) {<command call> <name $nm>}
Name [ [john (john smith)] {return(john_smith)}
      [mary (mary jones)] {return(mary_jones)}
      [ john (john brown)] {return(john_brown)}
      etc.
    ]
```

The word sequence "call john" can produce either of the following two interpretations:

```
{<command call> <name john_smith>}
{<command call> <name john_brown>}
```

In the case of ambiguous sentences, the natural language system produces all possible interpretations. (However, it will not produce interpretations that do not have the best possible score. For example, if one interpretation corresponds to a match of the entire sentence, the system will not produce interpretations that correspond to a match of only part of the sentence.)

Here are some further points to note:

- If a slot-filling command contains a variable that is not set, the command is not executed (and, hence, the slot does not get filled). For example, in the following grammar, the *source-account* slot does not get filled if the input is “to checking”:

```
Phrase +[ (from Account:source)
          (to Account:dest) ]
        {<source-account $source> <destination-account $dest>}
```

- Variables must be set before they are referenced. The following grammar will *not* cause the *source-account* slot to be filled:

```
Phrase ( from {<source-account $source>} Account:source )
```

- Slots are filled no more than once in a single interpretation. For example, the following interpretation will never be produced:

```
{<source-account savings> <source-account checking>}
```

- A grammar will not be matched to the input if that would cause a slot to be filled with two different values.
- An issue of precedence arises when you have a unary operator (?, * or +) and a command. For example, consider the following grammar:

```
Phrase ( ?please {<polite yes>} go )
```

The question is whether the *polite* slot gets set to *yes* when “please” is not present. If the command attaches to the construction ?please, then the slot will get filled; if it attaches to please, then the slot will not get filled.

The answer to the question is that the slot will *not* get filled, because the attachment of commands to constructions has precedence over the attachment of unary operators (such as ?) to their operands.

A SAMPLE APPLICATION

Here are the files that specify a sample banking application. These files can be found online in `$NUANCE/sample-packages`.

banking1.grammar

; Expressions like "my savings account", "checking" etc.

```
Account ( ?[my the]
    [ savings          {return(savings)}
      checking         {return(checking)}
      (money market)   {return(money_market)}
      (credit card)    {return(credit_card)}
      [(i r a) ira]    {return(ira)}
      mortgage         {return(mortgage)}
    ]
    ?account
  )
```

; Basic commands

```
Command [transfer      {<command-type transfer>}
        withdraw       {<command-type withdraw>}
        Balance        {<command-type balance>}
        Pay-bill       {<command-type pay_bill>}
      ]
```

; Ways to ask for a balance

```
Balance [ (what is [my the] balance)
          (tell me [my the] balance)
        ]
```

; Ways to specify bill payment

```
Pay-bill (pay ?for ?the Bill ?bill)
```

; Types of bill that can be paid

```
Bill [ phone          {<bill phone>}
      utilities       {<bill utilities>}
      (cable ?(t v))  {<bill cable>}
    ]
```

```

; An extremely limited number grammar

Number [ (five hundred)      {return(500)}
         (a thousand)       {return(1000)}
        ]

; The possible parameters of a banking command

Parameter [ (from Account:acct) {<source-account $acct>}
            (to Account:acct)  {<destination-account $acct>}
            (in Account:acct)  {<balance-account $acct>}
            (Number:amt [dollars bucks]) {<amount $amt>}
        ]

; A sentence is a command followed by zero or more
parameters

.Sentence (Command *Parameter)

```

banking1.slot_definitions

```

command-type
source-account
destination-account
balance-account
amount
bill

```

PROGRAMS

You can use the following programs for natural language processing.

Compilation Tools

The program *nuance-compile*, described in Chapter 1, will do all the compilation needed for runtime natural language understanding, provided there is a slot definitions file in the application directory. Of course, *nuance-compile* also produces all the files needed for speech recognition. If only interpretation of sentences is needed (i.e., recognition is not) it is quicker to use the program *nl-compile*. This program performs enough compilation for *nl-tool* or *parse-tool* to be run.

Sample usage:

```
% nl-compile $NUANCE/sample-packages/banking1
```

Runtime Tools

The program *nl-tool* lets you type sentences and see the interpretations that are produced for those sentences.

Sample usage:

```
% nl-tool -package $NUANCE/sample-packages/banking1
```

Sample execution:

```
> transfer five hundred dollars from checking to savings
Number of interpretations: 1
{<destination-account savings> <amount 500>
 <command-type transfer> <source-account checking>}
> ^D
```

The program *parse-tool* lets you type sentences and see whether they match a specified grammar (or any grammar). It will optionally show the parse tree for the sentence, if it matches.

Sample usage:

```
% parse-tool -package sample -print-trees
```

Sample execution:

```
the fish walks

.Sentence
  NounPhrase
    the
    fish
  VerbPhrase
    walks

fish walks

No parses
```


API FUNCTIONS

By default, when recognition is performed on an utterance, interpretation is performed on the recognition result. (You can control this behavior, however, with the parameter `rec.Interpret`. See Chapter 14 for information on Nuance parameters.) Your application must extract the natural language result from the recognition result, and then examine the components of the interpretation as desired. To do so, you will need to create an `NLResult` object, and free it when you are done.

The following API functions are provided to access the runtime component of the natural language understanding system. You need to include the file *nl.h* in your application code to call these functions, and link your program with an appropriate library (typically *librcapi.a*, but also possibly *libsapi.a*).

Initialization and Disposal Functions

The following functions create and dispose of the `NLResult` object:

```
NLResult *NLInitializeResult(NuanceStatus *status);

void NLFreeResult(NLResult *nl_result);
```

`NLInitializeResult()` creates and initializes an `NLResult` object. It should be called once at the beginning of the application.

`NLFreeResult()` frees the memory associated with an `NLResult` object.

Interpretation Functions

Typically, interpretation is performed whenever recognition is done, and you do not need to call a special API function for this purpose. However, if you need to perform interpretation on a sentence, the function `RCInterpret()` allows you to do so. The result of the interpretation is placed in the given `NLResult` object:

```
NuanceStatus RCInterpret(RecClient *rc, char *sentence,
                        char *grammar, NLResult *nl_result);
```

Interpretation Access Functions

The first step in accessing the interpretation of a recognition result is to get the `NLResult` object from the `RecResult` object. The function `RecResultNLResult()` allows you to do this. See the online documentation for information on `RecResult` objects.

In the case of an ambiguous sentence, the `NLResult` object will contain multiple interpretations. At any given point, exactly one interpretation is active. The slot access functions described on pages 52 through 54 apply to the active interpretation. You can set different interpretations to be active.

```
NuanceStatus RecResultNLResult(RecResult *rr,
                               int index, NLResult *nl_result);

NuanceStatus NLMakeIthInterpretationActive
    (NLResult *nl_result, int i);

int NLGetNumberOfInterpretations(NLResult *nl_result,
                                 NuanceStatus *status);
```

`RecResultNLResult()` fills the `NLResult` object you provide with the interpretations that were generated for the given recognition result. `index` is the index of the recognition result in which you are interested if you are using N-Best. (Otherwise, it should be zero.)

`NLMakeIthInterpretationActive()` makes the *i*th interpretation active. To look at all the interpretations, you can call this function with successively larger values of *i* until `NUANCE_ARGUMENT_OUT_OF_RANGE` is returned. The index of the first interpretation (which is also the initially active interpretation) is zero.

`NLGetNumberOfInterpretations()` returns the number of interpretations that were produced for the last processed sentence.

Score Access Functions

The natural language system may match only a subset of the words in the sentence against the given grammar. The score for an interpretation is simply the number of words matched in producing that interpretation. A higher score is better in that it indicates that the interpretation is more likely to be correct. All interpretations produced for a given sentence have the same score,

because the natural language system does not produce nonoptimal interpretations.

- **Note:** When the natural language system is used in conjunction with the speech recognition system (with the default `rec.BacktraceFinalsOnly=TRUE`), only perfect scores are possible.

Two functions allow you to check the score of the interpretations in a given `NLResult` object:

```
NuanceStatus NLGetInterpretationScore(NLResult *nl_result,
                                       int *num_words, int *num_phrases);

int NLIsscorePerfect(NLResult *nl_result,
                    NuanceStatus *status);
```

`NLGetInterpretationScore()` returns the score for the interpretation(s) that was (were) produced for the last processed sentence. The score is actually a pair of numbers: the number of words matched against the sentence and the number of phrases needed to perform that match. However, the number of phrases will always be one in the current version of the system, so you can safely ignore this parameter.

`NLIsscorePerfect()` indicates whether the interpretation(s) produced for the last processed sentence correspond(s) to a parse of the entire sentence by a single grammar. `NLIsscorePerfect()` provides a quicker way to check this than does `NLGetInterpretationScore()`.

Slot Access Functions

These functions allow you to access the filled slots in a given interpretation. Generally, you will call `NLGetIthSlotNameAndType()` repeatedly to get each slot name and the type of its value (usually, integer or string). Then, given the type, you can call either `NLGetIntSlotValue()` or `NLGetStringSlotValue()`, as appropriate, to get the value:

```
NuanceStatus NLGetIthSlotNameAndType(NLResult *nl_result,
                                       int i, char *slot_buffer, int slot_buffer_length,
                                       NLValueType *value_type);

NuanceStatus NLGetIntSlotValue(NLResult *nl_result,
                               char *slot_name, int *int_value);
```

```

NuanceStatus NLGetStringSlotValue(NLResult *nl_result,
    char *slot_name, char *buffer, int buffer_length);

NuanceStatus NLGetSlotType(NLResult *nl_result,
    char *slot_name, NLValueType *value_type);

int NLGetNumberOfFilledSlots(NLResult *nl_result,
    NuanceStatus *status);

NuanceStatus NLGetInterpretationString
    (NLResult *nl_result, char *buffer,
    int buffer_length);

```

Variables of type `NLValueType` have one of the following values:

```

NL_INT_VALUE
NL_STRING_VALUE
NL_STRUCTURE_VALUE
NL_LIST_VALUE

```

The types `NL_STRUCTURE_VALUE` and `NL_LIST_VALUE` and their corresponding API functions are discussed in Chapter 5.

Call `NLGetIthSlotNameAndType()` to get the name of the slot and type of the value from the *i*th filled slot of the currently active interpretation. The index of the first filled slot is zero. If you supply an index that is too large (e.g., you ask for the fifth slot when there are only four), `NUANCE_ARGUMENT_OUT_OF_RANGE` is returned. Thus, if you call this function with successively larger values of *i* until `NUANCE_ARGUMENT_OUT_OF_RANGE` is returned, then you can get all the slot names and types out of an interpretation. To get the actual slot value, you must call the appropriate type-specific access function. If `value_type` is set to `NL_INT_VALUE`, call `NLGetIntSlotValue()`; if `value_type` is set to `NL_STRING_VALUE`, call `NLGetStringSlotValue()`. See Chapter 5 for information on accessing slot values of type `NL_STRUCTURE_VALUE` or `NL_LIST_VALUE`.

Call `NLGetIntSlotValue()` to get the integer value of a specified slot and `NLGetStringSlotValue()` to get the string value of a specified slot.

Call `NLGetSlotType()` to get the type of the value of a slot. It is possible for slot to be filled with values of different types from one sentence to the next, so the result returned by this function is applicable only for a single

interpretation. By the same token, a type can be returned only if the slot is actually filled in the current interpretation.

Call `NLGetNumberOfFilledSlots()` to get the number of slots filled in the currently active interpretation.

`NLGetInterpretationString()` produces a string representation of the current interpretation, which is useful to display for debugging or demonstrations.

6

NUANCE SYSTEM ARCHITECTURE

In a speech recognition application, some functions are in nearly constant use, and some are required only intermittently. For instance, a speech application is almost always either playing (a prerecorded prompt, or text to speech) or recording (to save to disk, to send to the recognizer, or both). In contrast, a speech recognition application does not require constant use of the resources that provide recognition. It is sufficient to make connections to a recognition server on an as-needed basis.

The Nuance System was designed to meet these contrasting needs. Before you begin to program with the Nuance System, it is important to understand the relationships among its various parts. The main components of the Nuance System are:

- The Recognition Client (RecClient), which handles audio input and output and provides the APIs you use to build speech-enabled applications.
- The Recognition Server (RecServer), which provides recognition and interpretation servers for one or more RecClients.
- The Resource Manager, which manages connections between RecClients and RecServers, and performs load balancing to provide optimal throughput in medium and large client/server systems.

The Nuance System also provides the Dialog Builder to help ease application design and development. The Dialog Builder handles many of your

application's transactions with the RecClient transparently. For more information about the Dialog Builder, see Chapter 8.

In cases where you need more control or are creating a more specialized program, you can program directly to RecClient interfaces. These are described in Chapters 11- 13. The rest of this chapter describes the system architecture of the RecClient, RecServer, and Resource Manager.

RECLIENT

The RecClient performs audio and telephony functions, freeing applications to focus on higher-level issues such as call flow and dialog management. The RecClient is a process that runs in the background and can support one or more applications, running either on the same machine or remotely.

When you start an application, it must either start a new RecClient process or connect to an existing RecClient. The RecClient then attaches to the audio device for the application—for example, a telephone port—and provides the application with the following functionality:

- Acquiring digital audio data for recording and recognizing
- Playing prerecorded prompts
- Controlling telephone functions—notifying the application of incoming calls and remote hangups, dialing outbound calls, and so on
- Barge-in, which allows the caller to speak before the end of the prompt
- Energy-based endpointing to determine the beginning and end of the talker's speech
- Recognizing speech via dynamic connections to Nuance recognition servers
- Dual-tone multifrequency (DTMF) detection

The RecClient performs all these actions in the background, sending the application a message (via an event notification) whenever a change in state occurs.

The RecClient currently provides two sets of APIs you can use to invoke recognition features from your application:

- The RCEngine, a C++ class providing the functions needed to speech-enable an application.
- The Recognition Client API (RCAPI), a set of C functions for starting a RecClient and communicating with it. The RCEngine will eventually supersede this API, but it is currently still supported.

See "Choosing the Right API" on page 88 for information that will help you determine which API you should use to create your applications.

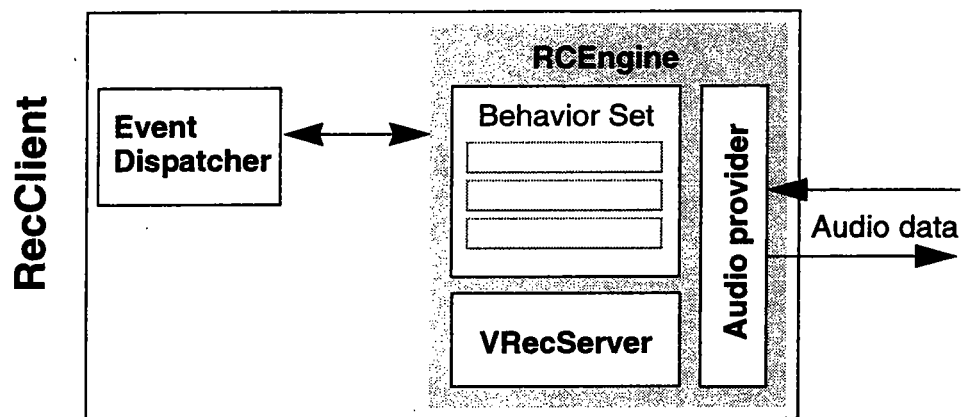
RecClient Components

Nuance Version 6 provides a new, C++ implementation of the RecClient that supports multiple channels, allowing a single RecClient to support more than one application. This implementation coexists with the RecClient API (RCAPI) provided in earlier releases—your programs that are built on the RCAPI can run with the new RecClient architecture, and take advantage of its support for multi-channel operation and DID management.

A RecClient has two major components:

- A *dispatcher*, which continuously checks for system events and calls the corresponding RecClient handling functions. You integrate your own event handling functions into a Nuance System by registering them with the dispatcher and associating them with specific events.
- One or more *RCEngine* objects, which provide functions for recognition, recording, playback, and call control, and that manage other RecClient components. The functions provided by this C++ interface are similar to the C functions in the RCAPI. Each RCEngine supports a single application, and contains:
 - A set of *behaviors*. Each behavior is an object that provides response mechanism for one or more types of events (for example, timeouts) the RCEngine might generate. You create additional behaviors and add them to the RCEngine to handle events as appropriate for your application. See "Handling Events with Behaviors" on page 179 for more information on behaviors and why and how to create them.

- A *VRecServer*, which provides an interface between the RecClient and the RecServer. You only need to work with this object if you want to access recognition services directly instead of calling RCEngine recognition functions. See Chapter 13, “The VRecServer API,” for information on programming with a VRecServer object.
- An *audio provider*, a set of C interfaces that handle the reception and transmission of live audio data to and from the current hardware. Audio providers are described in Chapter 15, “Nuance Audio and Telephony Providers.”



When you instantiate the RCEngine for your application, the RCEngine object handles construction of its own audio provider, VRecServer, and behavior set.

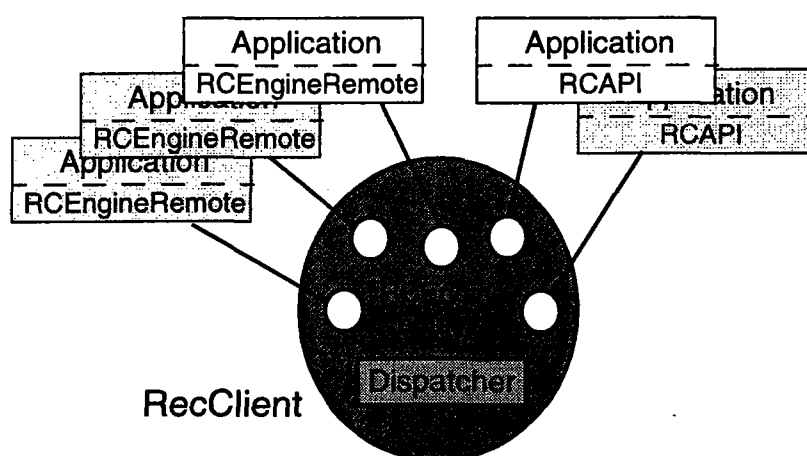
A RecClient might also contain a standalone VRecServer associated with the dispatcher. You might use a configuration like this, for example, if you want to work with pre-recorded data or want to invoke recognition services directly from your own audio interfaces.

The Multichannel RecClient

The RecClient can support multiple applications simultaneously by containing more than one RCEngine. The RecClient allows multiple RCEngine instances to coexist in a single thread, sharing a single dispatcher. This is useful in large configurations where process context-switching may be prohibitive. All RCEngine functions are thread-safe, so that you can call them from a different thread than the thread that is dispatching events.

The RecClient supports applications that are running both on the same machine and on different machines than the *recclient* process is running on. The Nuance System provides a special RCEngine implementation—the RCEngineRemote class—that you use as the basis for applications that may run remotely from the RecClient. RAPI-based applications can also connect to a RecClient running on another machine.

The multi-channel RecClient creates an RCEngine object to support each application as it connects, regardless of whether it is built on the RCEngineRemote or RAPI interface:



Starting a RecClient

Before starting any applications that need to connect to a multichannel RecClient, you need to start a *recclient* process. The parameters `config.RecClientHostname` and `config.RecClientPort` identify the location of the RecClient to connecting applications. If you start an RCEngineRemote-based application and it cannot find the specified RecClient, program initialization fails. If you start an RAPI-based application and it cannot find a RecClient, it starts its own dedicated RecClient as a child process.

See Chapter 7, “Starting the Nuance System,” for information on how to start a *recclient* process.

DID Management

Multi-channel RecClients also provide a DID manager that works with RCEngine objects using a Dialogic audio provider. You specify the incoming numbers for which each application can accept calls, and the DID manager

routes calls accordingly. See "Support for DID Management" on page 265 for information on how to set this up.

Audio Input

The RecClient supports several audio/telephony interfaces, called *audio providers*. The programming interface, however, is identical regardless of the telephone interface currently in use by the RecClient. This insulates your applications from the differences between audio interfaces, making your applications platform-independent. You can select the audio provider at runtime, so you don't need to recompile to use different audio providers.

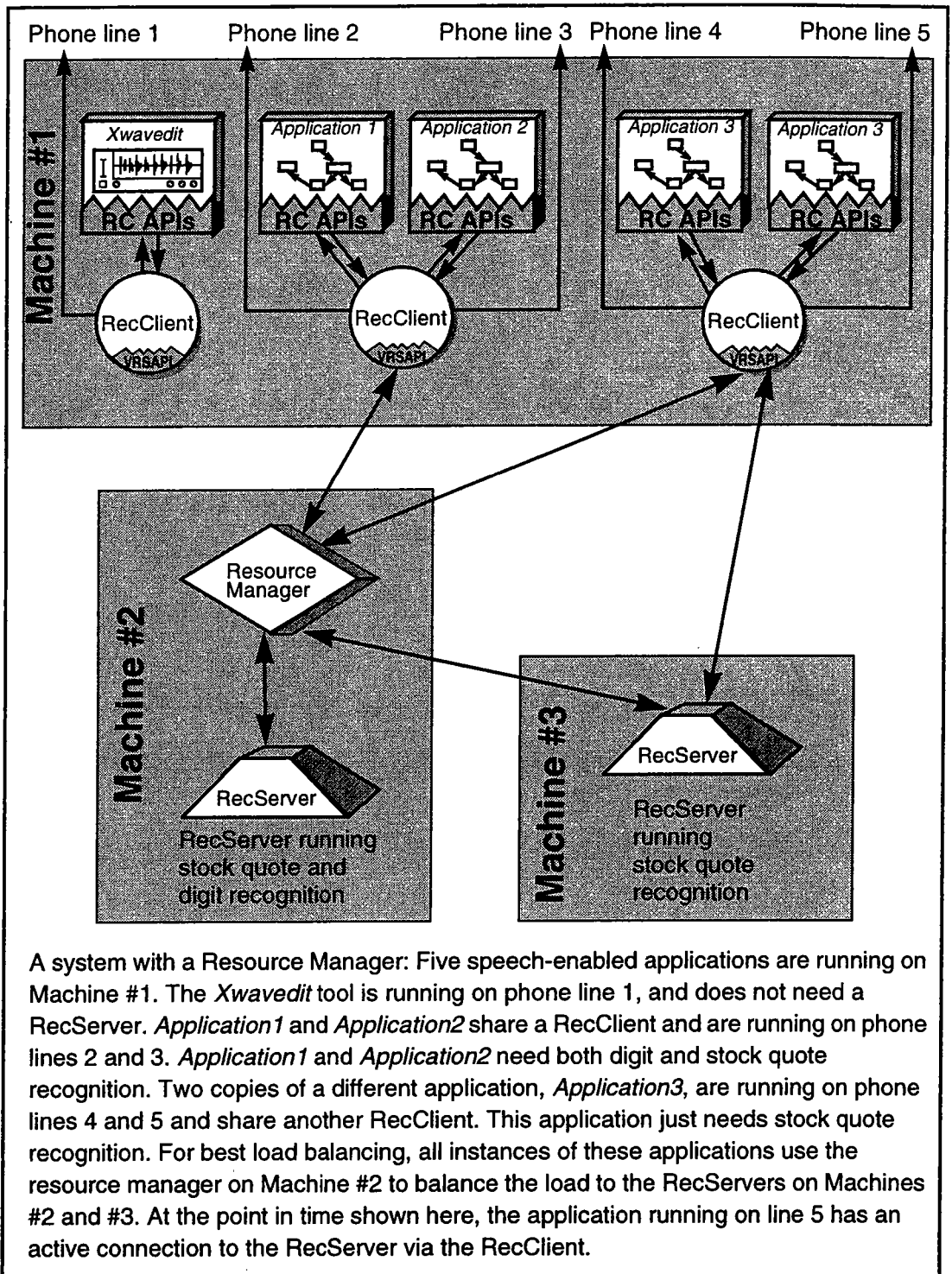
See Chapter 15 for information on the set of supported audio providers. You can also write your own audio providers to interface with unsupported audio devices.

Scalability

For larger systems where multiple telephone lines share several RecServers, you can configure RecClients to establish connections to RecServers through the Resource Manager. This usage allows optimal load balancing across the RecServers and dynamic updating of the system. This is the most powerful configuration of the Nuance System, suitable for large call center applications handling thousands of calls an hour.

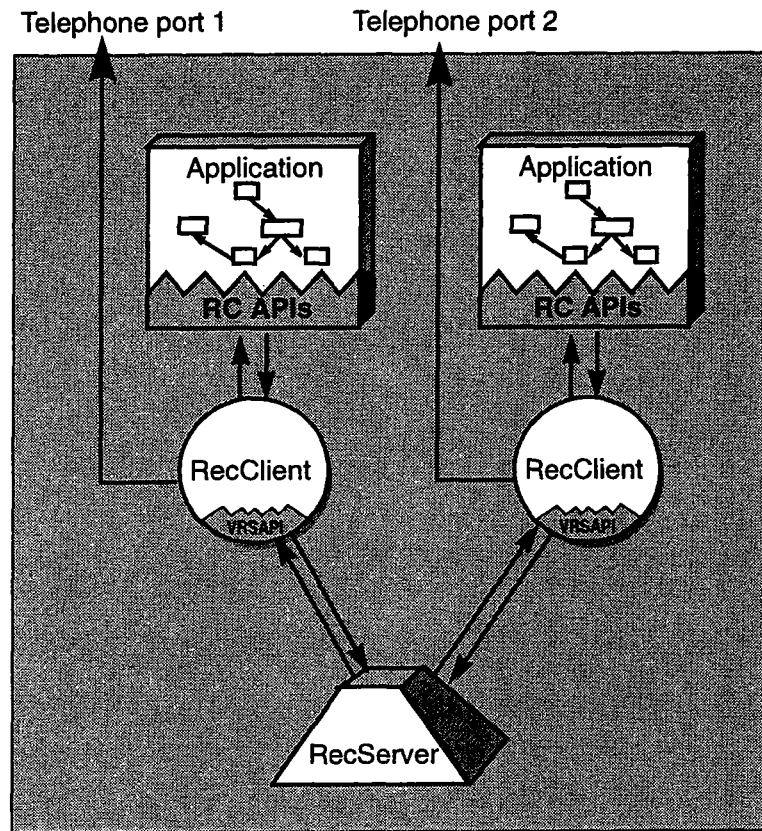
The following diagram shows an example:

Because a RecClient, possibly supporting multiple applications, can connect to multiple RecServers and a RecServer can receive connections from multiple RecClients, complex system architectures are possible. The Resource Manager provides load balancing for these systems.



For smaller configurations where only a single RecServer is needed, RecClients can establish dedicated connections directly to a RecServer that is already running.

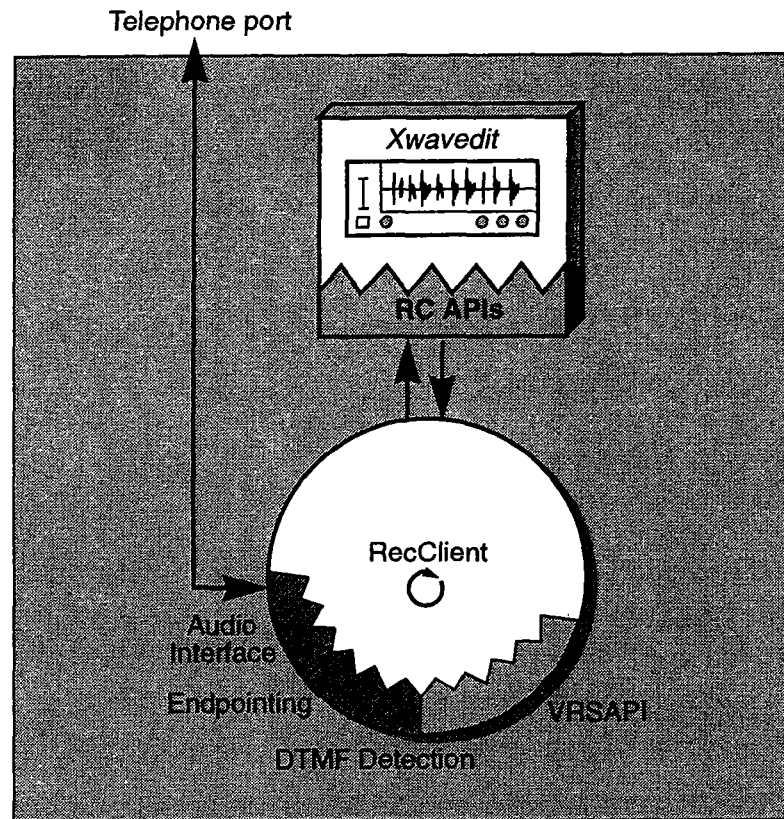
A small system:
Two instances of speech recognition applications each have their own RecClient to connect to a different telephone line, but the RecClients both use the same recognition server when recognition is needed. The RecClients shown here could support multiple applications.



For individual desktop configurations, the RecClient can start a RecServer on the local machine, effectively creating an entire client/server recognition system in the background. This configuration permits essentially no sharing of resources, but is suitable for individual use and for prototyping of new applications.

The RecClient can also run without connections to any recognition servers. In this mode, the RecClient can provide playback and recording, but not recognition. For example, the Nuance waveform editor, *Xwavedit*, works in this way, creating a RecClient that it uses for playback and recording, but not offering any recognition functionality. This diagram shows an example:

The *Xwavedit* tool calls RecClient programming interfaces to create and control a RecClient. This RecClient supports playback, recording, and telephony, but does not connect to a RecServer, and is not capable of performing recognition.



REC SERVER

Because a speech recognition application actually performs recognition only a certain percentage of the time, the RecServer was designed to be used intermittently by multiple RecClients running different applications. This allows the CPU and memory resources of the RecServer to be shared across multiple client applications, and provides the following advantages:

- *Distributed Architecture.* Recognition, which can be CPU-intensive, can be run on a different machine from the machine running the application and audio interface.
- *Scalability.* More RecServers can be added as needed as the volume increases.
- *Memory Savings.* Several recognition data objects, such as acoustic models and recognition grammars, can be efficiently loaded into a single

RecServer, and so can be shared across multiple recognition searches and multiple applications.

- *Fast Client Initialization.* Initialization of the recognition engine can take 20 seconds or more, but when a RecServer is started in advance, individual instances of a client application can connect to it and recognize speech in seconds.
- *CPU Sharing.* Because a given client application is connected to a RecServer only when it needs it, those RecServers can be used by other clients.
- *Support for Multiple CPU (SMP) Machines.* On those platforms that support threads, the RecServer can spawn as many threads as there are CPUs available.
- *Reliability.* The RecServer is designed as a two-process system, with one parent initializing all data objects and then starting a child process that handles all connections. If the child process ever fails, the parent immediately starts a new child in its place.
- *Audio Buffering.* The Nuance System architecture decouples audio acquisition from recognition processing, so the application that wants to recognize the caller can start *recording* at any time, even if the RecServer is busy. If the RecServer is completely busy, data can still be submitted for recognition, but is buffered until the RecServer is ready to process it.

RESOURCE MANAGER

In large systems where clients only use recognition servers intermittently, resources are used much more efficiently if recognition tasks are distributed evenly across the recognition servers. This is called *load balancing*. Nuance Version 6 introduces the Resource Manager to allow for optimal balancing of recognition load across a dynamic number of recognition servers.

When you include a Resource Manager in your configuration, all RecClients and RecServers connect to the Resource Manager, which then allocates servers for particular application requests. Connections to the Resource Manager are made dynamically, so, for example, you can shut down a RecServer with an old version of a grammar and start a RecServer with an updated grammar in

its place. RecClient connections are similarly dynamic, with RecClients connecting to the Resource Manager as they begin running.

The Resource Manager's primary responsibility is to identify the least loaded RecServer for every request from a RecClient. After connecting to a Resource Manager, a RecClient issues a request to it each time it needs recognition services. The manager monitors which RecServers are busy, which can satisfy requests for different grammars, and so on. When it receives a request, it identifies the best RecServer and notifies the RecClient. The RecClient then connects to that RecServer for the duration of that task (usually one sentence). The Nuance System was designed so that the negotiation between the client and manager takes a fraction of a second and occurs transparently to the application.

The Nuance System also optimizes reliability by allowing multiple Resource Managers to run in parallel. When you create a system configuration with multiple Resource Managers, all managers are fully functional—responding to all requests, monitoring all RecServers, and maintaining the same information. If any one of them fails, the other managers can fill in immediately with no degradation in system performance.

RECLIENT/RECSERVER CONNECTION MODES

Different configurations of the Nuance System and different applications make different kinds of demands on the RecServer. To allow for optimal performance in all settings, the Nuance System supports different types of connections between RecClients and RecServers.

The simplest connection mode is a direct, dedicated connection between a RecClient and RecServer. This is the style of connection used by individual and small configurations where no Resource Manager is in use. It provides good response for recognition queries, but does not allow for dynamic updates of the RecServer or for load balancing.

In systems using a Resource Manager, two connection modes are supported—a per-utterance connection or a per-call connection. The RecClient can select the mode requested based on the kind of application being run. Applications that do not use speaker-trained or caller-specific grammars achieve best

performance with a per-utterance connection. This mode provides the best load-balancing behavior.

In cases where repeated use of caller-specific grammars will be required, Nuance supports a per-call connection mode that connects a RecClient and RecServer for the duration of each speaker's call. This provides the best response for these applications, where speaker-trained grammars must be loaded into the RecServer, essentially creating a specific RecServer tuned for that speaker. Connections to other servers would need to duplicate that work, so the longer duration connection gives better performance.

CHOOSING THE RIGHT API

The Nuance System provides a number of APIs you can use as the basis of your application. This table will help you determine which API you should use for the type of application you want to create:

TABLE 7: NUANCE PROGRAMMING INTERFACES

API	Usage	Documentation
Dialog Builder	This high-level API lets you build a speech application based on a set of dialog states. Using this API lets you focus on the interface of your application, and provides you with recognition behavior that you can customize using Nuance parameters.	Chapter 8
RCAPI	Use this API to extend applications built using the Dialog Builder. This API will eventually be replaced by the RCEngine.	Chapter 9

TABLE 7: NUANCE PROGRAMMING INTERFACES

API	Usage	Documentation
RCEngine and RCEngineRemote	These C++ classes give you direct access to recognition client functionality including recognition, recording and playback, and call control. Use this API if you want a high level of control over the flow of events in your application or need to provide specialized event handling.	Chapter 12
VRecServer	This C++ class lets you send buffers of digital audio data directly to a network of RecServers. Use this interface if you are integrating the Nuance Recognition System with an existing IVR platform.	Chapter 13

8

DIALOG BUILDER

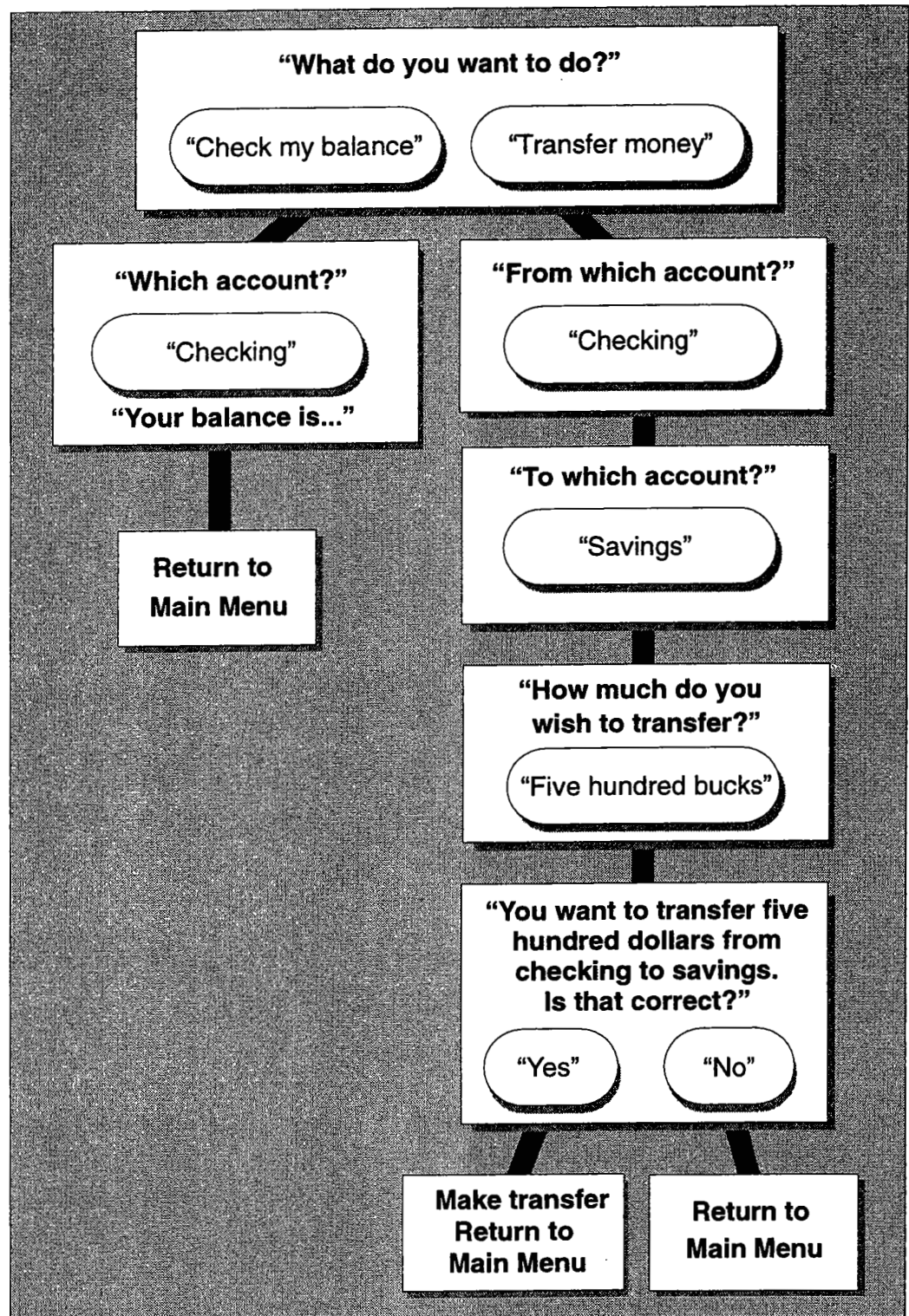
The Dialog Builder is a tool for building speech-enabled applications. It provides the highest-level API to Nuance System software. The applications you build with the Dialog Builder can also access the lower-level Recognition Client API described in Chapter 9. The Dialog Builder provides an alternative to building your application with the tools provided with your IVR platform.

To use the Dialog Builder, include *app.h* in your application code and link with *librcapi.so* and *libdialog.a* (*dialog.dll* on Windows platforms). You may want to use the sample banking application found in *\$NUANCE/sample-applications/banking/src* as a model. This directory also contains the Makefile and scripts you need to build and run a Dialog Builder application.

BASICS

To build an application with the Dialog Builder, you create a set of *dialog states*. A dialog state provides code that handles a single interchange between the user and the computer. For example, a banking application might have a state in which the application asks the user “How much do you want to transfer?” and the user responds with a dollar amount that the system would recognize. After a successful interchange, an application typically moves into a new dialog state—for example, the banking application might move into a confirmation state after the transfer amount state. You can envision the collection of interconnected dialog states in an application as a network.

An application is viewed as a network of dialog states.



An application can also have one or more state classes. A state class defines some sort of behavior that is shared among all the states in that class. For example, we might create a state class that specifies that in case of a recognition timeout, the user should be prompted with "Sorry, I didn't understand", and the state should be begun again.

State classes are organized into a hierarchy. That is, a state class A may have a parent class B, B may have a parent class C, and so forth up to a root class. A state inherits behaviors defined in all its ancestor classes.

THE APP OBJECT

Every Dialog Builder application creates an App object that stores all of the information needed by the application. To create an App object, call `AppNew()`:

```
App *AppNew(int *argc, char *argv[])
```

`AppNew()` should be called from your `main()` function and should pass in the command line arguments given to your application via `argc` and `argv`. The arguments used by `AppNew()` will be removed from `argv`, and `argc` will be decremented accordingly. Unused arguments remain intact so that your application can process them. At least one package must be specified (with `-package`) for `AppNew()` to succeed.

The arguments consumed by `AppNew()` include any Nuance parameters. In addition, `AppNew()` consumes the following arguments, which are specific to the Dialog Builder:

```
-prompt_path path
```

Specifies the directories to search for recorded prompts (see page 110).

```
-log_dir directory
```

Specifies that logging should be performed in the given directory (see page 114).

```
-text
```

```
-text_output
```

```
-text_input
```

```
-input_file file
```

Specifies text input or output, instead of audio (see page 114).

`-prompt_text file`

Specifies a file giving prompt transcriptions (see page 114).

`-monitor_port port`

`-monitor_host machine_name`

Specifies the Xmonitor connection (see page 115).

`-db_port port`

`-db_host machine_name`

`-db database`

Specifies the database connection (see page 120).

The App object contains several objects within it that you may need access to. The following functions provide this access:

Object	Accessor Function
RecClient	AppGetRecClient()
RecResult	AppGetRecResult()
NLResult	AppGetNLResult()
DBAnswer	AppGetDBAnswer()

The RecClient and RecResult objects are described in Chapter 9. The NLResult object is described in Chapter 4. The DBAnswer object is described later in this chapter.

DIALOG STATES

At the beginning of your application, you create the dialog states you need by calling `AppCreateState()`:

```
int AppCreateState(App *app, char *name, char *class,
                  AppFunction state_fn)
```

`AppCreateState()` creates a dialog state with the given name and stores it in the App object. It returns 1 if successful, and 0 otherwise. The *class* parameter specifies which class the state is a member of, NULL if none. The C

function, *state_fn*, defines what happens in the state. This function must take an App argument (and no other arguments), and return void. This function is referred to as the *state function*.

Here is an example of a state function for a state that gets a money amount from the user (lines are numbered for reference):

```

1. void HowMuchFn(App *app)
2. {
3.     NLResult *nl_result;
4.     NuanceStatus status;
5.     int dollars;
6.     char buf[100];
7.
8.     AppAppendPrompt(app, "how_much");
9.     if (! AppRecognize(app)) {
10.        return;
11.    }
12.    nl_result = AppGetNLResult(app);
13.    if (NLGetIntSlotValue(nl_result, "dollars", &dollars))
14.        == NUANCE_OK)
15.    {
16.        AppAppendPrompt(app, "you_specified");
17.        AppAppendNumberPrompt(app, dollars, 0);
18.        AppAppendPrompt(app, "dollars");
19.        AppGoto(app, "Date");
20.    }
21.    else if (NLGetStringSlotValue(nl_result, "help", buf, 100))
22.        == NUANCE_OK)
23.    {
24.        AppAppendPrompt(app, "help_how_much");
25.        AppGotoSelf(app);
26.    }
27. }
```

Line-by-line explanation follows:

- 8: *how_much.wav* is a file that contains a recording of the prompt "How much do you want to pay?" Notice that the prompt is only being *appended*. Appended prompts are not actually played until `AppRecognize()` is called.

-
- 9: AppRecognize() plays the appended prompts, starts recognition, and waits for recognition to finish. It returns 0 if the remainder of the state function should be skipped and 1 otherwise. (Later, we'll see why it might be desirable to skip the rest of the state function.)
 - 12: The `NLResult` object stored within the `App` object is automatically filled with the interpretation(s) for the recognition result.
 - 13-14: If the *dollars* slot is filled, we get its integer value.
 - 16-18: The system prompts something like "You specified fifty dollars".
 - 19: The system will go to the state called *Date* next (but only after this state is finished).
 - 21-22: If the *help* slot is filled...
 - 24: The system gives a help message appropriate to this state.
 - 25: Go back to the beginning of the current state once the current execution is finished.

The three most important API functions illustrated above are `AppGoto()`, `AppGotoSelf()`, and `AppRecognize()`:

```
void AppGoto(App *app, char *state_name)
void AppGotoSelf(App *app)
int AppRecognize(App *app)
```

`AppGoto()` and `AppGotoSelf()` set an internal parameter of the `App` object that specifies the next state to execute. After the current state function has finished, the next state's function will be called. `AppGoto()` or `AppGotoSelf()` should be executed once and only once per execution of a dialog state. If you call either of them more than once, only the first call has any effect.

`AppRecognize()` plays the stored prompts, starts recognition, waits for a recognition result, and checks for timeouts and other exceptional conditions.

It returns 0 if the rest of the state function should be skipped, and 1 otherwise. Therefore, it should generally be called as follows:

```
if (! AppRecognize(app)) {
    return;
}
```

The typical case in which `AppRecognize()` returns 0 is when there is no useful recognition result—for example, in the case of a timeout or reject.

Another useful function is `AppGo()`:

```
AppGo(App *app, char *initial_state)
```

`AppGo()` starts the application running. In general, it will loop indefinitely, starting one *session* after another. In a telephony environment, a session begins when a call is received, and ends when a hangup is detected or when `AppEndSession()` is called.

DIALOG STATE CLASSES

To create a dialog state class, call `AppCreateStateClass()`:

```
int AppCreateStateClass(App *app, char *name,
    char *parent, AppFunction entry_fn,
    AppFunction post_rec_fn)
```

`AppCreateStateClass()` creates a state class with the given name and stores it in the App object. It returns 1 if successful, and 0 otherwise. The parent parameter specifies the immediate parent of the new class, or NULL if none. Two C functions, *entry_fn* and *post_rec_fn*, define the behaviors associated with the class. These two functions must take an App argument (and no other arguments), and return void. They are referred to as the *entry* and *post-recognition* functions. The entry function typically sets parameters, while the post-recognition function usually processes exceptional conditions such as timeouts and rejects.

The entry function is executed when a state of that class is entered. The post-recognition function is executed immediately after recognition is performed in that state. The entry functions of high-level classes are executed before the entry functions of low-level classes. In contrast, the post-recognition functions of high-level classes are executed *after* the post-recognition functions of low-

level classes. Suppose we have a state that is a member of class *ChildClass*, and the parent of *ChildClass* is *ParentClass*. The order of events leading up to the execution of the state function is the following:

ParentClass's entry function

ChildClass's entry function

State function

The order of events inside `AppRecognize()` is the following:

Prompts are played

Recognition is performed

ChildClass's post-rec function

ParentClass's post-rec function

The entry function for *ParentClass* might be the following:

```
void ParentClassEntry(App *app)
{
    AppSetBeginSpeechTimeout(app, 5.0);
    AppSetEndSpeechTimeout(app, 5.0);
    AppSetEndRecognitionTimeout(app, 5.0);
    AppSetKeyTimeout(app, 2.0);
    AppSetBargeInAllowed(app, 1);
    AppSetKeyTerminators(app, "*");
    AppSetKeyMaxLength(app, 1);
    AppSetKeyTimeout(app, 1.0);
}
```

This entry function sets default values for various parameters. For example, we set the beginning-of-speech timeout to 5.0 seconds, turn barge-in on, and specify that the longest allowable sequence of touch-tone keys is 1. These parameters can all be overridden by specifying different values either within a lower class's entry function, or within a state function. See page 112 for more information on parameters.

The post-recognition function for *ParentClass* could be the following:

```

void ParentClassPostRec(App *app)
{
    char buf[100];
    NuanceStatus status;
    status = NLGetStringSlotValue(AppGetNLResult(app),
        "command", buf, 100);
    if (status == NUANCE_OK) {
        if (! strcmp(buf, "goodbye")) {
            AppEndSession(app);
        }
    }
}

```

This post-recognition function determines whether the command slot is filled with the value *goodbye*. If so, it hangs up the phone and ends the session. This behavior will be shared over all states in this class.

A more typical use of post-recognition functions is to handle exceptional conditions, such as rejects and timeouts, as discussed on page 108.

There is an additional subtlety of post-recognition functions. If any change of state function is called—such as `AppGoto()`, `AppGotoSelf()`, or `AppEndSession()`—anywhere within `AppRecognize()` (recall that the post-recognition functions are called from `AppRecognize()`), then `AppRecognize()` returns 0 (instead of the default 1). This means that the rest of the state function will not execute, assuming `AppRecognize()` is called as follows:

```

    if (! AppRecognize(app)) {
        return;
    }

```

So, in this example, if the command slot is filled with *goodbye*, we will not execute the second half of the state function, which checks slot values and transitions to a new state.

EXCEPTIONAL CONDITIONS

In the typical case, a user's utterance is recognized successfully, and an application continues on to the next state. However, several *exceptional conditions* can arise instead, as explained in Table 8.

TABLE 8: EXCEPTIONAL CONDITIONS

Condition	Description
Beginning-of-speech timeout	The user was silent for too long after the prompt finished.
End-of-speech timeout	The user spoke for too long.
End-of-recognition timeout	Too much delay occurred between when the user stopped speaking and the end of recognition.
Reject	The recognizer heard speech but could not determine what was said.
Unexpected key	A touch-tone key was pressed that was not allowed in the grammar.
Uninterpretable	The utterance could not be interpreted by the natural language system.
Session aborted	The session was aborted (e.g., the user hung up).

After `AppRecognize()` returns, you can call the function `AppGetUttStatus()` to see what exceptional condition arose, if any. This function may return the following status codes:

```
UTT_NORMAL
UTT_BEGIN_SPEECH_TIMEOUT
UTT_END_SPEECH_TIMEOUT
UTT_END_RECOGNITION_TIMEOUT
UTT_REJECT
UTT_UNEXPECTED_KEY
UTT_UNINTERPRETABLE
UTT_SESSION_ABORTED
```

Very often, the desired behavior in case of an exceptional condition is common across many states. For example, for a reject you may want to prompt the user

with "Sorry, I didn't understand", and start the current state over again. Thus, exceptional condition handling often happens in the post-recognition function associated with a state class, for example:

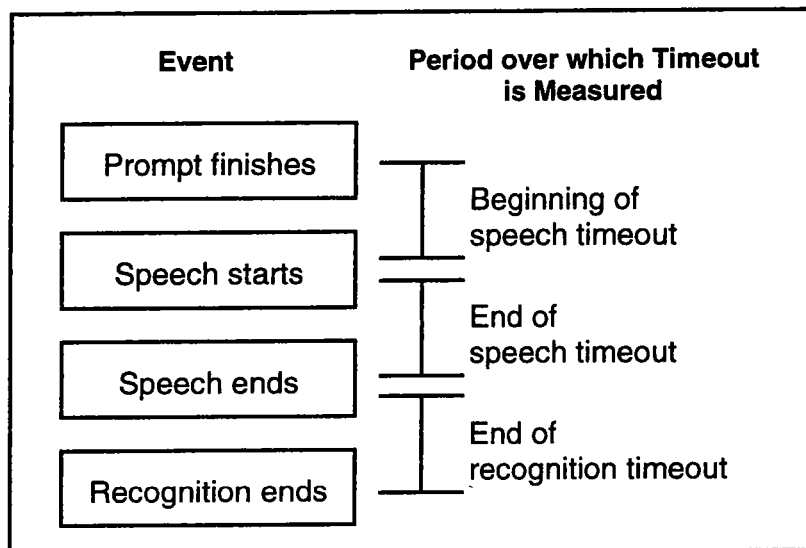
```
void Class1PostRec(App *app)
{
    UttStatus status = AppGetRecStatus(app);
    switch (status) {
        case UTT_BEGIN_SPEECH_TIMEOUT:
            AppAppendPrompt(app, "cant_hear_you");
            AppGotoSelf(app);
            break;
        case UTT_END_SPEECH_TIMEOUT:
        case UTT_END_RECOGNITION_TIMEOUT:
        case UTT_REJECT:
            AppAppendPrompt(app, "didnt_understand");
            AppGotoSelf(app);
            break;
        default:
            break;
    }
}
```

This function specifies that with a beginning-of-speech timeout, we tell the user we cannot hear him, while for an end-of-speech timeout, an end-of-recognition timeout, or a reject, we tell the user we did not understand him. In either case, we specify that the next state is the same as the current state.

Recall that calling a change-of-state function such as `AppGotoSelf()` causes `AppRecognize()` to return 0, which means that the second half of the state function will not execute, assuming your call to `AppRecognize()` looks like this:

```
if (! AppRecognize(app)) {
    return;
}
```

This figure diagrams the various timeouts in the usual sequence of events:



PROMPTING

Typically, in a state function you will append one or more prompts onto the end of a list maintained in the App object. Then, during the call to `AppRecognize()`, the prompts are played in the order in which they were appended. One function for appending prompts is `AppAppendPrompt()`:

```
NuanceStatus AppAppendPrompt(App *app,
                               char const *prompt_name)
```

Typically, *prompt_name* is the name of a file that contains a recorded prompt (minus the *.wav* suffix). The system looks in the directories specified in the prompt path, and appends the prompt if it can locate the file. The prompt path consists of the contents of the `PROMPT_PATH` environment variable, followed by `$NUANCE/data/prompts`, followed by the current directory. `$NUANCE/data/prompts` contains many useful recorded prompts, including numbers, letters, and date elements such as month names and days of the week. You can add to the beginning or end of the path by using functions such as the following:

```
NuanceStatus AppAppendToPromptPath(App *app,
                                     char const *path)
```

```
NuanceStatus AppPrependToPromptPath(App *app,
                                     char const *path)
```

There are also functions that make it easy to generate various types of complex prompts, for example,

```
NuanceStatus AppAppendNumberPrompt(App *app, int number,
                                    int id_mode)

NuanceStatus AppAppendDigitsPrompt(App *app, int number,
                                    int num_digits)

NuanceStatus AppAppendOrdinalPrompt(App *app, int number)

NuanceStatus AppAppendPhoneNumberPrompt(App *app,
                                         char const *number)

NuanceStatus AppAppendMoneyAmountPrompt(App *app,
                                         int dollars, int cents)

NuanceStatus AppAppendDatePrompt(App *app, char *month,
                                  int day, int year, char *modifier, char *day_of_week)

NuanceStatus AppAppendTimePrompt(App *app, int hour,
                                  int minutes, int pm)
```

The constituent prompts needed by all these functions exist in *\$NUANCE/data/prompts*.

Typically, you will not need to play the prompts that are appended to the prompter (that will be done automatically in `AppRecognize()`). But if you do need to, you can call the following function:

```
NuanceStatus AppPlayPrompts(App *app, int block)
```

If `block` is 0, the function returns immediately. Otherwise, it does not return until the prompts are finished playing.

Additional prompting functions exist but are not described here. See the prompting section of the header file *app.h* for details. See page 114 for information on how to get text prompts rather than speech prompts.

For recording prompts, you may want to use *Xwavedit*. You can crop each prompt with *Xwavedit*, and then normalize your prompts with *wavnorm*.

STATE PARAMETERS

Various parameters such as timeout lengths and rejection threshold control the behavior of your application in a particular state. These parameters are referred to as *state* parameters to distinguish them from Nuance parameters (e.g., `rec.Pruning`, `config.DebugLevel`). State parameters must be set upon each entry to a state, either in a state class's entry function, or in the beginning of the state function. They will be reset to default values after the state function completes.

In some cases, state parameters do much the same thing as a Nuance parameter (e.g., set the rejection threshold). If the proper parameter value varies depending on the dialog state, it is best to use the state parameter. In some cases, you might also have to set a Nuance parameter when the application is run.

Remember that you can set Nuance parameters on the command line (assuming that your main function passes *argc* and *argv* into `AppNew()`). You can also modify the `RecClient` directly (it can be gotten by calling `AppGetRecClient()`).

The following table lists the various state parameters:

TABLE 9: STATE PARAMETERS

Parameter	Get/Set Functions	Description
Grammar	<code>AppGetGrammar()</code> <code>AppSetGrammar()</code>	The recognition grammar.
Beginning-of-speech timeout	<code>AppGetBeginSpeechTimeout()</code> <code>AppSetBeginSpeechTimeout()</code>	The beginning-of-speech timeout (the length of time in seconds permitted between the end of the prompt and the beginning of the user's utterance).
End-of-speech timeout	<code>AppGetEndSpeechTimeout()</code> <code>AppSetEndSpeechTimeout()</code>	The end-of-speech timeout (the length of time in seconds permitted between the beginning of the user's utterance and the end).

TABLE 9: STATE PARAMETERS

Parameter	Get/Set Functions	Description
End recognition timeout	AppGetEndRecognitionTimeout() AppSetEndRecognitionTimeout()	The length of time in seconds permitted between the end of the user's utterance and the end of recognition.
Barge-in allowed	AppGetBargeInAllowed() AppSetBargeInAllowed()	Whether barge-in is currently allowed. The parameter <code>client.AllowBargeIn</code> must also be set to TRUE for barge-in to be possible.
Touch-tone terminators	AppGetTerminatorsKeys() AppSetTerminatorsKeys()	The set of keys that terminate a touch-tone key sequence.
Touch-tone maximum length	AppGetMaxKeyLength() AppSetMaxKeyLength()	The maximum length a sequence of touch-tone keys may be.
Touch-tone timeout	AppGetKeyTimeout() AppSetKeyTimeout()	The length in seconds allowed after a key press before another key press.
Rejection threshold	AppGetRejectionThreshold() AppSetRejectionThreshold()	The confidence score level that serves as the rejection threshold. Hypotheses with a confidence score below this level are rejected. For no rejection, this value is set to -1.

LOGGING AND DEBUGGING

The Dialog Builder offers various logging and debugging capabilities.

Logging

If you pass the argument `-log_dir directory` to `AppNew()`, useful information about the progress of a running application is logged in the given directory. A hierarchy of log directories is maintained, which groups logs by year, month, and day.

Logging does not cause a user's utterances to be recorded to disk. To do that, you must also set the parameter `client.WriteWaveforms` to `TRUE`.

See *app.h* for information on more logging functions.

Debugging

Various facilities are available to assist in debugging an application. One is simply to start your application with the Nuance parameter `config.DebugLevel=N` where *N* is some number between 1 and 4. A higher number leads to more debugging output.

Text Input and Output

For debugging, it can be useful to type to your application (instead of speaking) and to see text responses. To specify text input, start your application with the argument `-text_input`. Alternatively, to take input from a file, use the argument `-input_file filename`.

To specify text output, start your application with the argument `-text_output`. However, for this to work, it is necessary that the application have access to transcriptions of the prompts. By using *Xwavedit*, you can associate such transcriptions with your prompts. Alternatively, you can create a prompt text file, on which each line lists a prompt name and the text for that prompt, for example,

```
how_much      "How much do you want to transfer?"
from_account  "Which account do you want to transfer from?"
etc.
```

You can specify a prompt text file with the argument `-prompt_text filename`.

If the system cannot determine the transcription for a given prompt, it uses the prompt filename (without the directory path or the *.wav* suffix) as the transcription. For example, if your prompt is stored in */home/foobar/how_much.wav*, the fallback transcription is *how_much*.

The `-text` argument means to use both text input and output (it is equivalent to `-text_input -text_output`). When text input and output are both in effect, the application does not spawn a *RecClient*, and there is no need for a *RecServer* to be running to connect to. This can simplify and speed application startup.

Xmonitor

unix On Unix platforms, an X windows program is available to monitor a running application. It displays the recognized words and the interpretation, and, optionally, the SQL query and the database answer, if you are using the Nuance interface to a relational database (see page 120).

To start *Xmonitor*, specify a package and a port number:

```
Xmonitor -package sample -port 23456
```

If you want to display database queries and answers, add `-db`.

When you start the application, use the arguments `-monitor_host` and `-monitor_port` to specify the machine on which you ran *Xmonitor* and the port you specified to it.

MISCELLANEOUS FEATURES

This section describes several supplementary features that simplify or speed up application development.

Subdialogs

The normal way for moving among dialog states is via `AppGoto()`. As the name suggests, this is analogous to a *goto* in a programming language like C. However, another possibility is to call to a subdialog from within a state; this is analogous to a function call in C.

The API functions related to calling subdialogs are the following:

```
void AppCall(App *app, char *state_name, void *argument)
```

Calls the subdialog beginning at the given state, passing in the given argument. The subdialog is exited upon a call to `AppReturn()`.

```
void AppReturn(App *app)
```

Tells the application to end the current subdialog.

```
void *AppGetArgument(App *app)
```

Gets the argument specified in the last call to `AppCall()`.

Unlike `AppGoto()`, which returns immediately, `AppCall()` actually invokes the subdialog and does not return until after it encounters a call to `AppReturn()`. You may not call `AppGoto()` before you call `AppCall()` (although you may do so afterwards).

Callbacks

You can specify several types of callback functions, described in Table 10, which will be executed when certain conditions are met. The most useful of these is the “begin session” callback in which you can initialize application-specific data that must be reinitialized for each phone call.

TABLE 10: CALLBACKS

Callback	When It Is Executed
Begin session	At the beginning of a session (i.e., a phone call)
End session	At the end of a session
Begin speech	When the start of an utterance is detected (or when the first touch-tone key is pressed)
End speech	When the end of an utterance is detected
Begin prompt	Just before prompt playback begins
End prompt	When prompt playback finishes (which will not occur if the user barges in)

TABLE 10: CALLBACKS

Callback	When It Is Executed
Begin recognition	Just before recognition begins (when the prompt begins if barge-in is enabled; just after the prompt finishes otherwise)
End recognition	When recognition finishes (or a timeout of some sort occurs)

The API functions for setting these callbacks are the following:

```
void AppSetBeginSessionCallback(App *app,
    AppFunction callback)

void AppSetEndSessionCallback(App *app,
    AppFunction callback)

void AppSetBeginSpeechCallback(App *app,
    AppFunction callback)

void AppSetEndSpeechCallback(App *app,
    AppFunction callback)

void AppSetBeginPromptCallback(App *app,
    AppFunction callback)

void AppSetEndPromptCallback(App *app,
    AppFunction callback)

void AppSetBeginRecognitionCallback(App *app,
    AppFunction callback)

void AppSetEndRecognitionCallback(App *app,
    AppFunction callback)
```

Personal Data

You can store data items that are specific to your application in the App object. To do this, call `AppSetPersonalItem()` to store an item in the App object under a given name. To get the object out, call `AppGetPersonalItem()`.

```
void AppSetPersonalItem(App *app, char *item_name,
    void *item_value)
```

Sets the personal item with the given name to the given value.

```
void *AppGetPersonalItem(App *app, char *item_name,
int *success)
```

Returns the value of the personal item with the given name. *success is set to 0 if there is no item with that name and to 1 otherwise. The success parameter may be NULL if you do not want to check its value.

Entry Conditions

Consider a dialog state in which the user is queried “How much do you wish to transfer?”, and which accepts “help” as a response, in addition to a money amount. If the user requests help, the system says “Please speak the amount of money you wish to transfer” and begins the state again. If nothing else is done, the user is prompted again with “How much do you wish to transfer?”, which is redundant given the help message.

What we would like is a way to condition the behavior of a state depending on how we got there. Entry conditions provide you with a way of doing this. You specify an entry condition for the next state when you execute an `AppGotoCond()`. In the next dialog state, the entry condition is checked and the appropriate behavior is performed.

The relevant API functions are the following:

```
void AppGotoCond(App *app, char *state_name,
int next_entry_condition)
```

Specifies the next state to go to and the entry condition upon entering that state.

```
void AppGotoSelfCond(App *app, int next_entry_cond)
```

Specifies that the next state to go to is the current state and that the entry condition is `next_entry_cond`.

```
void AppCallCond(App *app, char *state_name,
void *argument, int next_entry_condition)
```

Calls the subdialog beginning at the given state and specifies the entry condition for that state.

```
int AppGetEntryCondition(App *app)
```

Gets the current entry condition.

In the example above, there would be at least two calls to `AppGotoCond()`. To transition to the transfer amount state from the previous state, we would call:

```
AppGotoCond(app, "transfer_amount", 0)
```

To transition back to the beginning of the transfer amount state after a help request, we would call:

```
AppGotoCond(app, "transfer_amount", 1)
```

Then, at the beginning of the transfer amount state function, we would have:

```
if (AppGetEntryCondition(app)) {
    /* Do nothing - do not play prompt */
}
else {
    AppAppendPrompt(app, "how_much_transfer");
}
```

Note that `AppGoto()`, `AppCall()`, and `AppGotoSelf()` specify an entry condition of 0.

Contextualization

Contextualization refers to the process of updating something to reflect the context of utterance. The function `ContextualizeDate()` updates a date value to reflect the time at which it was uttered. For example, if the date value corresponds to *today*, the function sets the day, day_of_week, month, and year features to their current values. See the header file *app.h* for more information on `ContextualizeDate()`.

The function `UpdateCumulativeInterpretation()` updates a cumulative interpretation by adding the filled slots in `inc_result` to those in `cum_result`:

```
NuanceStatus UpdateCumulativeInterpretation
(NLResult *cum_result, NLResult *inc_result)
```

Slots that are already filled in `cum_result` have their values replaced if they are also filled in `inc_result`. This is a useful way of keeping track of the slots filled over the course of a dialog.

INTERFACES TO EXTERNAL SOFTWARE

Various facilities are used for interfacing to external (non-Nuance) software. Two types of external software are supported: database systems and text-to-speech systems.

Databases

Two database systems are currently supported: Informix and Mini SQL. These are both relational database systems that process SQL queries. Mini SQL is less powerful than Informix, but also more affordable. You can download Mini SQL from the Hughes Technology web site <http://Hughes.com.au>.

API

If you pass the argument `-db database_name` into `AppNew()`, the system will automatically connect to the given database (and disconnect when the App object is deleted). To execute a query, call `AppExecuteQuery()`:

```
NuanceStatus AppExecuteQuery(App *app, char *query)
```

The database result is stored in a `DBAnswer` object, which is stored within the App object. You can access this result by calling `AppGetDBAnswer()`:

```
DBAnswer *AppGetDBAnswer(App *app)
```

You can create your own `DBAnswer` object with `DBNewAnswer()`.

If you need to save a database answer, you can initialize a second `DBAnswer` object with `DBNewAnswer()` and then use `DBCpyAnswer()` to copy the answer into this new object:

```
DBAnswer *DBNewAnswer(NuanceStatus *status)
```

```
NuanceStatus DBCpyAnswer(DBAnswer *from, DBAnswer *to)
```

Several functions get data out of a `DBAnswer` object. Here are some of the most useful ones. See the header file *db-interface.h* for more information.

```
int DBGetNumberOfRows(DBAnswer *answer)
```

```
NuanceStatus DBGetIntColumnValue(DBAnswer *answer,
    int row_index, char *column, int *column_value)
```

```
NuanceStatus DBGetFloatColumnValue(DBAnswer *answer,
    int row_index, char *column, double *column_value)
```

```
NuanceStatus DBGetStringColumnValue(DBAnswer *answer,
    int row_index, char *column, char *buffer,
    int buffer_length)
```

Installation

To interface with either the Informix or Mini SQL relational database systems, you must install that software, and link two additional things with your application:

- The library or libraries provided with the database that enable a C interface. In the case of Informix, this requires you to purchase its E-SQL product.
- The appropriate Nuance interface object file, either *msql-interface.o* or *informix-interface.o*, which can be found in *\$NUANCE/lib/\$MACHINE_TYPE*.

The interface object file should appear on your link line before any other Nuance libraries (e.g., *librcapi.so*). The database libraries can appear anywhere after the interface object file. So, for example, your link line might look something like this (irrelevant details omitted):

```
gcc -o ../bin/sparc-solaris/myapp ../obj/sparc-solaris/
myapp.o /home/nuance/v6.0/lib/sparc-solaris/msql-
interface.o -lrcapi -ldialog /home/msql/msql-1.0.16/
targets/Solaris-2.4-Sparc/lib /libmsql.a
```

Text-to-Speech

unix The only text-to-speech system that Nuance has an interface to is Entropic's TrueTalk. However, you may be able to call other systems directly from your application code. Contact Nuance technical support for more information.

API

The most common way to interface to the Entropic system is through `AppAppendPrompt()`, which was described on page 110. If `AppAppendPrompt()` cannot find a recorded *.wav* file, it will try to determine the transcription of the prompt, and use text-to-speech to generate the prompt, if Entropic is installed. The system looks for transcriptions in a prompt text file. On each line the prompt text file lists a prompt name and the transcription for that prompt, for example,

```

how_much      "How much do you want to transfer?"
from_account  "Which account do you want to transfer from?"
etc.

```

For the application to be able to find the prompt text file, you must start your application with the argument `-prompt_text filename`.

You can also use `AppAppendTTSPrompt()` to generate text-to-speech output:

```
NuanceStatus AppAppendTTSPrompt(App *app, char const *text)
```

The advantage of using `AppAppendPrompt()` is that if you later record a `.wav` file for the prompt, your application will automatically start using that, rather than text-to-speech.

Installation

To use Entropic, you must purchase Entropic's Developer TrueTalk product.

Once you have Entropic installed, you can use it in your application by linking two things:

- The Entropic libraries provided with Developer TrueTalk
- Nuance's Entropic interface object file, *entropic-interface.o*, which can be found in `$NUANCE/lib/$MACHINE_TYPE`.

The Entropic interface object file should appear on your link line before any other Nuance libraries (e.g., *librcapi.so*). The Entropic libraries can appear anywhere after this object file. So, for example, your link line might look something like the following (irrelevant details omitted):

```

gcc -o ../bin/sparc-solaris/myapp ../obj/sparc-solaris/
myapp.o /home/nuance/v6.0/lib/sparc-solaris/entropic-
interface.o -lrcapi -ldialog /home/entropic/lib/libttd.a /
home/entropic/lib /libwavesynt.a /home/entropic/lib/
libsolaris.a

```

Finally, when you run your application, you should set `client.UseTTS` to `TRUE` and `client.TTSHost`, `client.TTSDataport`, and `client.TTSCommandPort` to the appropriate values, depending on how and where you started the text-to-speech server. (The data and command port parameters may not need to be set.) See the Entropic documentation for information on these values, and on running the text-to-speech server.

9

PROGRAMMING WITH THE RECCLIENT API

The Nuance Dialog Builder is constructed on top of an interface called the RecClient API, or RAPI. This library of event-driven functions can be compiled into your application, transforming a C language application into a speech recognition client.

Any program or application that incorporates speech recognition is considered a speech recognition client—that is, a program that makes use of speech recognition services. The Nuance System comprises a recognition server (the *recserver*) and a recognition client. The recognition client provided with the Nuance System processes live audio data read from one of the supported audio devices (see Chapter 15 for more information). With audio access taken care of, you are free to concentrate on the workings of the application, without having to deal with the routine transfer of digitized audio from the audio device to the server. For applications that need to recognize audio originating from a source that Nuance does not support, you may need to implement a custom *audio provider* (see Chapter 15). However, most developers use the RecClient API with the existing audio providers to write their speech recognition applications.

- The functionality provided by the RAPI is now also provided by the C++ class RCEngine, described in Chapter 12. Nuance intends that the Dialog Builder will be ported to the RCEngine, and that the RCEngine APIs will eventually replace the RAPI. Both APIs are currently supported, however,

for new development that is not based on the Dialog Builder, use of the RCEngine interfaces is recommended.

THE RAPI LIBRARY

On SPARC Solaris and i386 Solaris platforms, the RAPI library is shipped as a shared object named *librcapi.so*. On other UNIX platforms, the library is shipped as a static library named *librcapi.a*. On Windows, the library is shipped as a dll named *rcapi.dll*.

- On UNIX platforms, the script *\$NUANCE/SETUP* adds *\$NUANCE/lib/\$MACHINE_TYPE* to your *\$LD_LIBRARY_PATH* environment variable. On Windows, your *PATH* variable is set when you install the Nuance System. This allows the loader to find the RAPI library at runtime.

RAPI VERSIONS

With the freedom of version independence comes greater responsibility. The behavior of the Nuance System may vary slightly from version to version. If you want to write a version-independent application, you may need to be able to query, at runtime, the version of the library with which you are dynamically linked.

For this purpose, three version-checking functions are available. The header *nuance-version.h* includes these three functions:

`const char *NuanceVersionName()` returns a string indicating the name of the Nuance version—for example, “v6.0”.

`int NuanceVersionNumber()` returns an integer indicating the number of the Nuance version. Nuance version 6.2.3 would be indicated by the integer 623.

`const char *NuanceVersionDate()` returns the date, as a string, on which this version of the Nuance System was compiled.

In addition, *rcapi.h* includes the function `int RAPIVersionNumber()`. This function currently returns 1, which is for forward compatibility. Future versions of the RAPI may return 2, 3, and so forth and be documented accordingly.

- Note: No RecClient pointer is required, so this function may be called before `RCInitialize()`.

FUNCTIONS AND CODES

The RecClient API (RCAPI) is an event-driven interface, in which most functions are nonblocking, and many initiate background actions that continue after the function returns. The background process (the RecClient) notifies the application of its progress by sending it software *events*. The event-driven programming paradigm is detailed in Chapter 10.

All Nuance System API functions return a `NuanceStatus` code, either as the return value or by reference. You should always compare this code to `NUANCE_OK` to ensure that the function returned successfully. You can convert a `NuanceStatus` code into a string with the function `NuanceStatusMessage()`. For example, `NuanceStatus : :NUANCE_OK` is converted into the string "NUANCE_OK."

The principles of using most RecClient API functions are explained here. For full details on each function, see the online documentation.

LEARNING BY EXAMPLE

The best way to learn the RecClient API is through an example. Two sample applications, including source code, are included with the Nuance System installation. These are a text-based recognition application called *sample-application*, whose source code is in `$NUANCE/src`, and a graphical recognition application, *Xapp*, whose source code is in `$NUANCE/Xapp/src`. The two most important files for understanding *Xapp* are *xapp-defs.c* and *xapp-callbacks.c*. Many developers begin to write applications by modifying one of these two programs, and much can be learned by studying them.

SETUP

To use the RecClient API, your application should include the following header file, which contains prototypes for all the RecClient API functions:

```
#include "rcapi.h"
```

This file is found in the directory `$NUANCE/include`. To call the RecClient API functions, you must link your application with the library `librcapi.a` (`librcapi.so` on Solaris or `rcapi.dll` on Windows), which is located in the directory `$NUANCE/lib/$MACHINE_TYPE`. See `$NUANCE/src/sample-application.c` and `$NUANCE/src/Makefile` for examples of use.

INITIALIZATION

In any recognition application, the first RecClient API function called is `RCInitialize()`, which instantiates a new recognition client object and initializes it based on a specified recognition configuration. Like most RecClient API functions, `RCInitialize()` is nonblocking. It returns immediately, but initialization of the RecClient process continues in the background. When background initialization completes, the RecClient notifies the application by sending it the `NUANCE_EVENT_INIT_COMPLETE` software event.

`RCInitialize()` returns a pointer to a RecClient object or NULL if an immediate error occurred. This object encapsulates the recognition system, and is required as the first argument to all other RecClient API functions.

The purpose of `RCInitialize()` is to initialize a recognition system and configure it for a certain recognition task, as specified in a recognition package. A recognition package comprises one or more recognition grammars that have been combined with the developer's choice of acoustic models. Chapter 1 describes the specification and compilation of recognition packages.

One way to invoke `RCInitialize()` is as follows:

```
NuanceConfig *config;
NuanceStatus status;
RecClient *rec_client;
config = NuanceConfigBuild("/home/disk1/mypackage",
    &status);
if (!config) {
    /* handle the error, and exit */
}

rec_client = RCInitialize(config, &status);
if (!rec_client) {
```

```

        /* handle the error, and exit */
    }

```

The function `NuanceConfigBuild()` loads the contents of the recognition package `/home/disk1/mypackage` into a `NuanceConfig` object. This object is then passed to `RCInitialize()`, which reads its contents and begins initialization.

Of course, a flexible application would most likely read the name of the recognition package from the command line. The `NuanceConfig` API offers a function `NuanceConfigBuildFromCommandLine()`, which will read one or more packages from the command line for you. The advantage of using this function is that it will also read other Nuance System arguments from the command line, and add them to the `NuanceConfig` object. The result is that by the addition of a single line, your application will accept any Nuance System argument on the command line. Here is an example:

```

NuanceConfig *config;
NuanceStatus status;
RecClient *rec_client;
config = NuanceConfigBuildFromCommandLine(&argc, argv, 1,
    &status);
    if (!config) {
/* handle the error, and exit */
    }

    rec_client = RCInitialize(config, &status);
    if (!rec_client) {
        /* handle the error, and exit */
    }

```

`NuanceConfigBuildFromCommandLine()` looks for the command line argument(s) `-package packagename`. It also reads any other arguments that look like Nuance System parameters, and removes them from the command line. In this way, your application will be as configurable as any Nuance System application. For more details, see the online documentation for `NuanceConfigBuildFromCommandLine()`. Nuance System parameters are fully explained in Chapter 14.

Note that in the example, when initialization is complete, the `RecClient` object `rec_client` will be capable of recognition, playback, and recording. Most of the initialization time is associated with recognition. If your

application does not make use of recognition, but needs only to record and play back, there is a faster way to initialize. If you pass a zero as the third argument to `NuanceConfigBuildFromCommandLine()`, the function will not require any recognition packages to be specified on the command line, and if `RCInitialize()` is passed a `NuanceConfig` object that does not contain any recognition packages, it will not initialize recognition functionality. The resulting initialization will take only a few seconds. If an application does not initialize recognition but attempts to call any recognition functions, they will return the error `NUANCE_NOT_CONNECTED`.

- Note: If, indeed, your application does require recognition functionality, the `RecClient` API first looks for a recognition server that is already running on the machine and port specified by the values of `config.ServerHostname` and `config.ServerPort`. For each recognition package specified, if a server running that package is found, the `RecClient` connects to that server instead of creating its own, and initialization then takes only a few seconds. It is very simple to start a recognition server (see page 93), and running such a server on your machine can speed up your application-development cycle. Existing servers will allow your application to start in only a few seconds, whereas letting the application start them could take as long as a few minutes, depending on the size and scope of the grammar(s) in each package. However, if you make a change to one of the grammars, you must restart that *recserver* that is responsible for it.

Completion of Initialization

Your application cannot call any recognition, playback, or recording functions when initialization is taking place in the background. It must wait for the Nuance System event `NUANCE_EVENT_INIT_COMPLETE` to occur. Event handling is explained in detail in Chapter 10. The simplest way to handle this event is to simply wait for it, as demonstrated here:

```
NuanceStatus fn_status, init_status;
NuanceEvent event = NUAANCE_EVENT_INIT_COMPLETE;
fn_status = RCWaitForNextEvent(RecObj, &event, 120,
    &init_status, sizeof(init_status));
```

This function waits up to 120 seconds for any event to arrive. When `RCWaitForNextEvent()` returns, `fn_status` indicates whether

an event actually occurred (NUANCE_OK) or the function timed out (NUANCE_TIMED_OUT). If an event occurs, the event variable contains the type of event that was generated.¹ If event equals NUANCE_EVENT_INIT_COMPLETE, then init_status indicates whether the background initialization task completed successfully. If init_status equals NUANCE_OK, it is safe to proceed to other RecClient API calls.

REGISTERING CALLBACKS

Once initialization has completed successfully, the application can register *callbacks*. Callbacks are functions that the developer is requesting the Nuance System to call when certain events occur. For example, all applications should register a callback for the event NUANCE_EVENT_PROCESS_DIED. This event indicates that one of the Nuance System subprocesses died unexpectedly. Although this rarely occurs, robust applications will want to know about it:

```
void notify_me_that_a_process_died(void *user_arg,
    NuanceEvent ev, void *evdata)
{
    NuanceStatus status = *(NuanceStatus *)evdata;

    fprintf(stderr, "Subprocess died because '%s';
        Quitting.\n", NuanceStatusMessage(status));
    exit(EXIT_FAILURE);
}

main()
{
    ...

    status = RCRegisterCallback(RecObj,
        NUANCE_EVENT_PROCESS_DIED,
        notify_me_that_a_process_died, NULL);
```

¹Currently, the only event other than NUANCE_EVENT_INIT_COMPLETE that can occur is NUANCE_EVENT_INCOMING_CALL. Most applications can probably exit with an error if this event occurs, since they cannot call any other RCAPI functions until initialization has completed. In the more general case, it is common for several events to be expected in any order. Applications that wait for events must be prepared to handle any of the events that can occur at a given time. See the discussion of *Waiting for Events* on page 145.

```
...
}
```

In this example, the developer has requested that whenever the event `NUANCE_EVENT_PROCESS_DIED` occurs, the Nuance System should call the function `notify_me_that_a_process_died()`. This function will not actually be called until the application calls `RCProcessEvents()` or one of the `RCWaitFor...()` functions, as that is the only time the `RecClient` event handler gets control.

The Nuance System has eleven events, described in detail in Chapter 10. Some applications will register callbacks for all the events or some subset of the events, and some only for `NUANCE_EVENT_PROCESS_DIED`. Callbacks need to be registered only once per type of event (not for each occurrence of the event). Callback registration should take place after initialization, but before any recognition, playback, or recording actions are taken.

RECOGNITION

The `RecClient` API provides two mechanisms for recognizing an utterance. In the first, the system uses spectral energy to identify the beginning and end of speech. One function, `RCRecognize()`, puts the process in motion. As with other `RecClient` API functions, `RCRecognize()` returns immediately. When the talker begins to speak, the system detects the onset of speech and sends the event `NUANCE_EVENT_START_OF_SPEECH` to the application. When the talker has finished speaking, `NUANCE_EVENT_END_OF_SPEECH` occurs. The developer may choose to register callbacks for these two events. Finally, when the recognition engine reaches its conclusion, it sends the event `NUANCE_EVENT_FINAL_RESULT`, which is accompanied by a `RecResult` structure.

The `RecResult` structure contains the final recognition result, the `NLResult`, and several recognition statistics, including the confidence score of the recognition result. If N-best recognition was requested (by running the server with the parameter `rec.DoNBest=TRUE`), the `RecResult` structure may contain more than one recognition hypothesis. See the online documentation for `RecResultNumAnswers()`, `RecResultString()`, `RecResultNLResult()`, and associated functions for details on extracting recognition strings and other information from the `RecResult` structure.

- Note: N-Best recognition returns a ranked list of possible recognized strings, but requires additional memory to do so. See the short topic on N-Best on page 346 for more information.

The RecClient API also allows the developer to specify the start and end of the utterance to be recognized, bypassing the spectral energy endpointing used by `RCRecognize()`. This capability may be used in combination with a push-to-talk, release-to-stop button, and is appropriate in noisy environments where automatic endpointing may not work. In this case, the developer can call `RCStartRecognizing()` to start sending audio data to the recognition server, and `RCStopRecognizing()` to stop. Only the audio data acquired in the interval between these two function calls will be sent to the server for processing. When `RCStartRecognizing()` and `RCStopRecognizing()` are used, the two events `NUANCE_EVENT_START_OF_SPEECH` and `NUANCE_EVENT_END_OF_SPEECH` do not occur, but the final result still arrives with the event `NUANCE_EVENT_FINAL_RESULT`.

Recognition from File

The application can perform recognition from file by using `RCRecognizeFile()`. This function behaves almost identically to `RCRecognize()`, except that it takes its input from the given file instead of using the live audio device. It also takes an additional parameter, which determines whether to endpoint the file before sending the data to the recognizer. You cannot perform recognition on a file at the same time as you are recognizing live audio, because the RecClient has only one recognition resource available at any given time. `RCRecognizeFile()` is useful for re-recognizing an utterance using a different recognition grammar or different parameter settings.

Partial Recognition Results

As recognition progresses, the Nuance System is capable of periodically sending the application its “best guess” so far. These best guesses are called *partial results*. Applications may simply want to display partial results, since they make demonstrations more interesting, or, they can be

used to make decisions on the fly.² In either case, if you register a callback for the event `NUANCE_EVENT_PARTIAL_RESULT`, partial results will arrive periodically (with the period specified by the parameter `rec.PartialResultSeconds`), each carrying a `RecResult` structure containing the partial result.

- ❖ **Caution:** Partial results are only best guesses. Subsequent partial results may contradict earlier ones, and the final result may also contradict partial results.

Data Collection

Developers may want to collect recognized utterances from live use of an application. These may be used to verify recognition accuracy, or as a test set to improve the accuracy of the system by tuning. The Nuance System automatically saves recognized waveforms to disk if the parameter `client.WriteWaveforms` is set to `TRUE`. For more information on this and related parameters, see Chapter 14.

RECORDING

Sometimes applications will want to record an utterance without actually recognizing it. The functions `RCRecord()`, `RCStartRecording()`, and `RCStopRecording()` are provided for this purpose. They are identical to the recognition functions, except that they take as an argument the name of the file to be saved, and they do not send the recorded audio to the *recserver*.

²For instance, an application that asks the user to say a credit card number might want to allow pauses of several seconds between sets of digits without cutting off the talker, but would prefer a snappy cutoff when the talker is finished with the sixteenth digit. The parameter `ep.EndSeconds` controls the amount of silence allowable inside an utterance, and can be set on the fly. So, an application could set `ep.EndSeconds` to 3 seconds at the start of the credit card utterance. When the talker pauses after the fourth, eighth, and twelfth digits, the system will not cut the talker off. Then, when partial results indicate that the talker is up to the thirteenth digit, the application could set `ep.EndSeconds` to 0.5 second, ensuring a snappy response at the end of the utterance.

PLAYBACK

The RecClient API provides two functions that play audio out through the audio device: `RCPlayFile()` and `RCPlayLastUtterance()`. `RCPlayFile()` takes as an argument a character string listing one or more audio files, separated by spaces, which should be played out the audio device. The files are concatenated and played as a unit. When the final audio samples have been played, the event `NUANCE_EVENT_PLAYBACK_DONE` is sent to the application.

`RCPlayLastUtterance()` plays back the last utterance that was recorded or recognized. This function is very helpful for verifying audio quality. It, too, generates the event `NUANCE_EVENT_PLAYBACK_DONE` when all samples have been played.

To avoid a race condition, a play command cannot be issued while playback is in progress. Before another playback command is issued, the application must wait for `NUANCE_EVENT_PLAYBACK_DONE` to occur. Otherwise, the second command returns the error `NUANCE_PLAYBACK_BUSY`.

- Note: You will usually wait for `NUANCE_EVENT_PLAYBACK_DONE`, but it is also possible to register a callback function for it. This allows the application to perform other tasks and to be notified automatically when the playback is complete.

SIMULTANEOUS PLAYBACK AND RECORDING/RECOGNITION

Recording or recognizing while playback is in progress is possible if the client. `AllowBargeIn` parameter is set to `TRUE` for RecClient objects that use a telephony-based audio provider. The correct sequence of calls to allow the user to barge in over the prompt is to call `RCPlayFile()` followed by `RCRecord()` or `RCRecognize()`.

By default, if the user then barges in over the prompt, no `NUANCE_EVENT_PLAYBACK_DONE` is generated, because playback is killed internally. Set the parameter `client.KillPlaybackOnBargein` to `FALSE` if you want prompt playback to complete even when the user barges in.

Barge-in is covered in detail on page 331.

WAITING FOR EVENTS

It is sometimes convenient to wait for a certain NuanceEvent. Programmers should be aware that there are very few cases in which the type of event that will occur next can be predicted. For instance, if the application starts recognition, NUANCE_EVENT_START_OF_SPEECH will usually be the next event. However, if the caller says nothing and RCRecognize() times out, the next event will instead be a NUANCE_EVENT_FINAL_RESULT with a result of "<timeout>". Similarly, if the application plays a prompt and waits for NUANCE_EVENT_PLAYBACK_DONE, the next event might instead be NUANCE_EVENT_REMOTE_HANGUP, indicating that the caller hung up the phone. Since the sequence of events cannot be predicted, Nuance Communications recommends that applications be written in an event-driven manner, relying on callbacks to drive the flow of the application. However, when it is necessary to wait for a certain event to occur, the application must be sure to handle all possible events that may occur at that time.

There are several ways to wait for one or more events to occur. The following example illustrates the use of RCWaitForNextEvent():

```
NuanceStatus fn_status;
int start_sample[2];
NuanceEvent event = NUANCE_EVENT_START_OF_SPEECH;

fn_status = RCWaitForNextEvent(RecObj, &event, 120.0,
    start_sample, sizeof(start_sample));
```

This function waits up to 120 seconds for the next event to arrive. When RCWaitForNextEvent() returns, fn_status indicates whether an event actually occurred (NUANCE_OK) or the function timed out (NUANCE_TIMED_OUT). If an event occurred, event indicates the type of event that was generated. If the event that occurred was NUANCE_EVENT_START_OF_SPEECH, then start_sample[0] and start_sample[1] will contain two indices describing when start of speech occurred.³ If any other event occurs, start_sample will be untouched.

³See Chapter 10 for more details on the data associated with this and other events.

RecClient TELEPHONY

The Nuance System RecClient supports several telephone interfaces, including Dialogic. To simplify the connection to these telephone interfaces, the RCAPi offers integrated call-control functionality. RCAPi functions exist to detect incoming calls, answer calls, place outgoing calls, detect DTMF, and so forth. These function calls are identical regardless of the audio interface, so that applications using the Nuance RCAPi are hardware-independent. In fact, the Nuance parameter `audio.Provider` allows you to switch between audio interfaces via a command-line argument.

Since the RecClient allows you to select an audio interface at runtime, your application needs to be aware of the difference between *telephony providers* and *nontelephony providers*. Applications that use telephony-based audio providers are required to perform call control in addition to playback and recognition. To find out whether the RecClient is using a telephony-based audio provider, call the function `RCQueryTelephonySupport()`. It will tell you whether or not your application needs to do call control.

The RCAPi provides call-control functionality by maintaining an internal state machine that keeps track of what telephony state it is currently in. The initial state is *idle*, at which time no playback or recording/recognition can take place. When a connection to a remote telephone device has been established, the RecClient transitions to its *in-progress* state, at which time playback and recording/recognition are possible. Transitioning between the two states is described in more detail in the following sections.

Detecting an Incoming Call

When an incoming call is detected, the RecClient sends the event `NUANCE_EVENT_INCOMING_CALL` to the application. The associated data item that accompanies the event is a structure containing two strings: the ID of the intended recipient and the caller ID. This allows the application to choose the calls it wants to accept. Often, neither of these two pieces of information is available, because the telephony device does not support it or the phone company has disabled the feature.

Answering an Incoming Call

When the application receives an incoming call event, it can choose to answer the call. To do this, call `RCAnswerCall()` with the phone number of the incoming call's remote ID (caller ID). If the application does not care about the remote ID, or if caller ID was not indicated with `NUANCE_EVENT_INCOMING_CALL`, it can call `RCAnswerCall()` with the constant string `RC_ANY_CALLER`. This will answer any incoming call that has already been detected, regardless of remote ID. Although this function blocks until the connection is made or an error occurs, it returns almost instantaneously. If it returns successfully (i.e., with a `NUANCE_OK` status code), the call is now in progress, and the application can perform playback and recording/recognition. If the remote caller has hung up since the incoming call event was generated, this function returns `NUANCE_TELEPHONY_CHANNEL_CLOSED`. This function can be called only when the `RecClient` is in the idle state. If successful, the `RecClient` will then be in the in-progress state.

Placing an Outgoing Call

To place an outgoing call from within your application, use the function `RCPlaceCall()` with the phone number you want the `RecClient` to dial. You can call this function only in the idle state. As with most other `RCAPI` functions, this function returns immediately, but puts a background process in motion that eventually generates the event `NUANCE_OUTGOING_CALL_COMPLETED` with an associated status code. If the connection is completed—that is, the called party picks up the phone—a call is in progress, and the application can now perform playback, recording, and recognition.

Detecting Completion of an Outgoing Call

When the event `NUANCE_EVENT_OUTGOING_CALL_COMPLETED` occurs, it is the asynchronous response to the application's call to one of two `RCAPI` functions: `RCPlaceCall()` or `RCTransferCall()`. The application must know which of these two functions generated the event. A `NuanceStatus` accompanies this event to indicate whether the command completed successfully. If `RCPlaceCall()` is called and this event is generated, a status

of NUANCE_OK means that the connection has been established. The RecClient state is now in progress. If a value of NUANCE_TELEPHONY_REMOTE_BUSY accompanies the event, the number dialed is already in use, and the RecClient object remains in the idle state. If RCTransferCall() is called and this event is generated, a status of NUANCE_OK means the transfer completed successfully, and the telephone line connected to the application is now available for other calls. In this case, the RecClient object returns to the idle state. If a value of NUANCE_TELEPHONY_REMOTE_BUSY accompanies the event, the number to transfer to is already in use, and the RecClient object remains in its in-progress state.

Transferring a Call to Another Number

Once a call is in progress, the application can choose to transfer it to another number by calling RCTransferCall(). This function returns immediately and eventually generates a NUANCE_OUTGOING_CALL_COMPLETED event with an associated status code. If the transfer completes successfully, the application's telephone channel is available for new calls, and the RecClient object returns to its idle state. Transferring is usually done by putting the original (in-progress) call on hold, dialing the new number, and finally connecting the original call with the new one, which disconnects the application from both of the two calls.

Be aware that most audio providers implement only a *blind transfer*, which means that they transfer the call without checking to see whether the number to transfer to is already in use (busy). A more sophisticated provider will implement a *smart transfer*, monitoring the new call's progress and reestablishing the original call if the number to transfer to is busy. This can usually be done only on a digital line (T1, ISDN), because the busy detection is done via a digital signal on the line rather than through signal processing by the telephony device.

The application should not rely on a specific type of underlying transfer implementation, however. In the case of a blind transfer, the application will always appear to have successfully transferred the call, possibly leaving the original caller with a busy signal. The smart transfer may return an unsuccessful transfer status to the application, but the original call will still be connected with the application, so it can, for example, play back a prompt

reporting that the number to transfer to is busy, which allows the caller to try other numbers or perform other application-specific tasks.

- Some audio providers or telephone line configurations may not support call transfer.

Determining Whether a Call Is in Progress

The Boolean Nuance configuration parameter `audio.CallInProgress` can be queried with `RCGetIntParameter()` to determine whether a call is currently in progress. Be aware that the value of this parameter could already be out of date by the time that the application receives it if, for example, the remote person hangs up.

Detecting Remote Hangup

When the in-progress telephone connection is terminated remotely, a `NUANCE_EVENT_REMOTE_HANGUP` is generated. The data accompanying this event form a null-terminated string indicating the remote ID that generated the hangup. This information is not always available. When this event occurs, the `RecClient` object returns to its idle state. The application is free to make or receive new calls.

Detecting DTMF Tones

The `RecClient` generates a `NUANCE_EVENT_DTMF` whenever a DTMF tone is detected. The application can choose to use or ignore these events. The accompanying data is a single ASCII character indicating the DTMF tone that was detected. DTMF tones can be detected only while a call is in progress. To restrict the set of DTMF tones that the application will be notified of, you can use the `dtmf.Mask` parameter. Be aware that not all telephony-based audio providers have their own DTMF detection mechanisms. In this case, the `RecClient` performs the signal processing necessary to detect the tones. However, this means that DTMF can be detected only between a call to `record/recognize` and the end-of-speech event. The *cfone* and *dialogic* audio providers, however, do perform continuous DTMF detection, so this restriction does not apply in those cases.

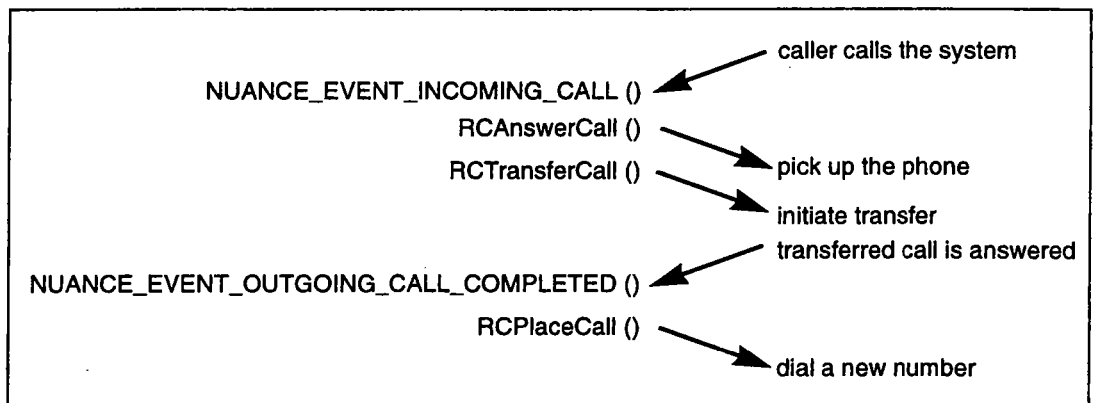
Disconnecting a Call

To disconnect a call in progress, call `RCHangup()`. This hangs up the call and returns the `RecClient` to the idle state. This function can be called in any `RecClient` state. If it is already in the idle state, it remains in that state. Therefore, calling it repeatedly will not adversely affect the application or the `RecClient`. If `RCHangup()` is called while a call is being transferred, the new outgoing call to be transferred to is disconnected, and the application reconnects to the call in progress. Calling `RCHangup()` a second time disconnects the original call as described.

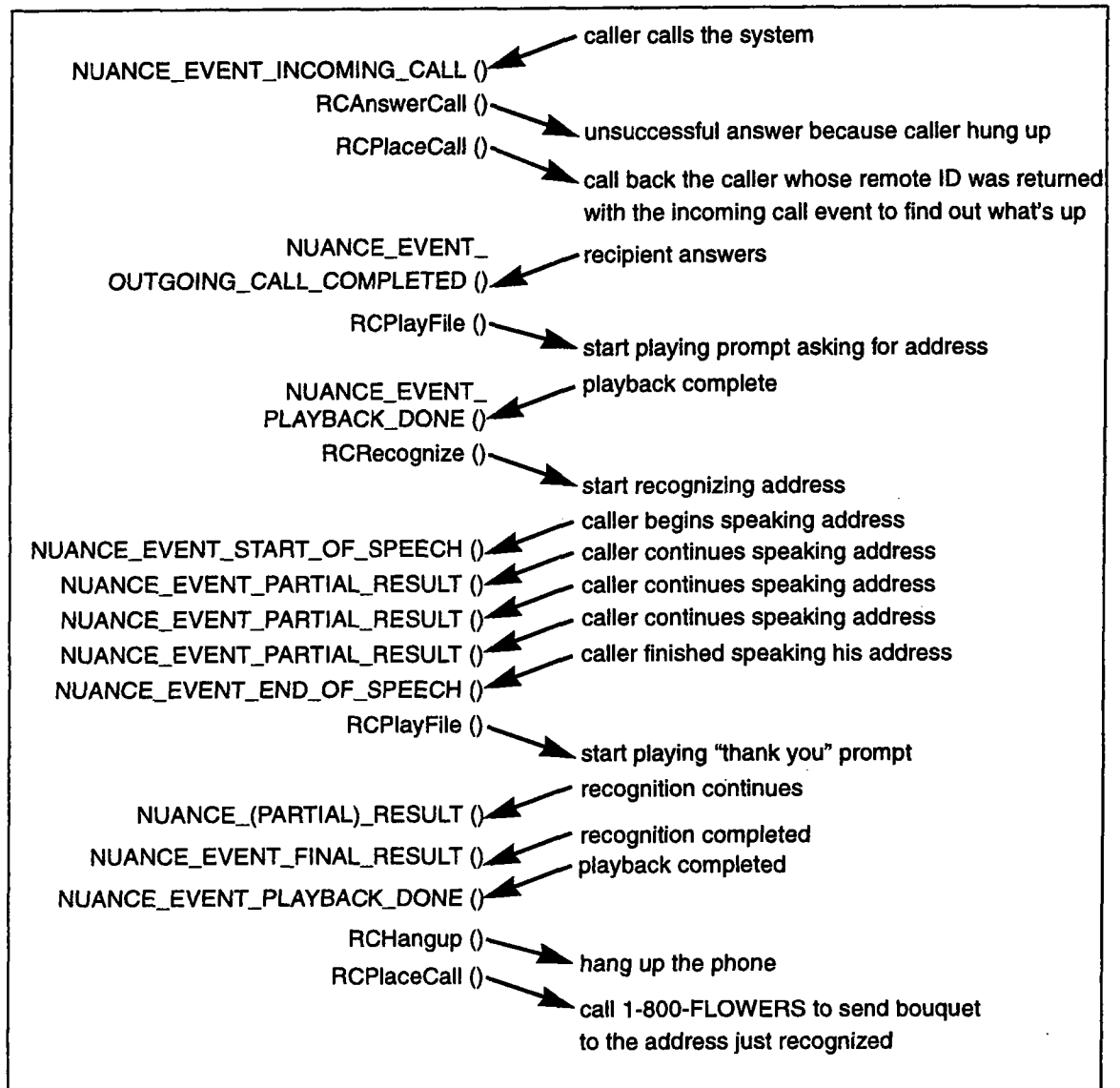
EXAMPLES OF RECCLIENT APPLICATION CALL FLOWS

The following diagrams illustrate several possible interactions between an application making RCAPI function calls, and a person at the other end of the phone.

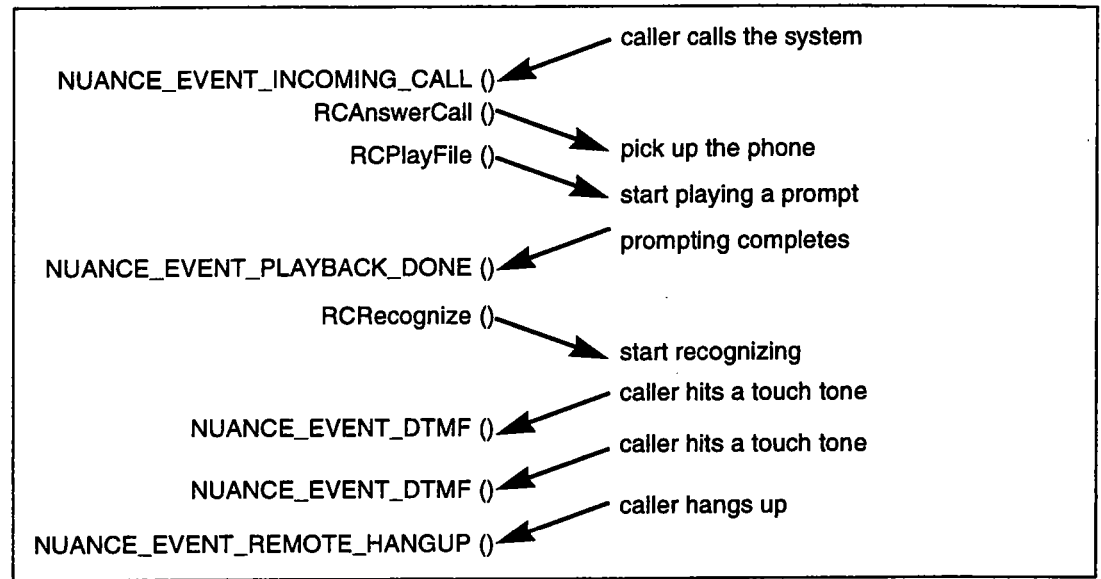
In this interaction, the application answers an incoming call, transfers that call to another number, and then is free to place another call.



In this interaction, the application tries to answer an incoming call, but not before the caller hangs up. The application uses caller ID to call the person back, asks for an address, and sends flowers.



In this interaction, the application detects and answers a call, plays a prompt, and starts recognition. The caller presses two DTMF keys, and then hangs up.



OTHER FEATURES

The RecClient API has two other significant features: the ability to request natural language interpretation, and support for dynamic grammars.

Natural Language

If the developer specifies natural language commands as part of the recognition grammar, then recognition results will arrive with embedded natural language results already included. In these cases, the application does not need to explicitly request natural language interpretation. However, an application may want to use the natural language engine to parse some text generated via a different mechanism. To do this, call `RCInterpret()`, which blocks and returns the `NLResult` generated by parsing the specified text with the given grammar.

Dynamic Grammars

To dynamically insert a set of words into a recognition grammar at a prespecified position, call `RCSetWordGrammar()`. This function transmits the specified `WordGrammarInfoList` to the recognition server and blocks (typically much less than 1 second) until the server has finished updating its

grammars. See Chapter 16 for more information about dynamic grammar modification.

DELETING THE RECCLIENT OBJECT

When the application is done using the RecClient, it should call `RCTerminate()`. This will ensure a clean disconnect from the recognition server and free up any system resources, including audio/telephony devices.

MORE INFORMATION

Each RCAPI function is covered in more detail in the online documentation.

14

MODIFYING THE NUANCE SYSTEM WITH NUANCE PARAMETERS

The Nuance Speech Recognition System is designed to be highly configurable. It has a large set of *Nuance parameters* that you can set to configure the RecServer, the RecClient, or your application, or that you can query to learn more about how the Nuance System is working. These parameters give you a high degree of control over every aspect of the Nuance System. You can set many Nuance parameters from the command line, allowing you to experiment with changes in your application or the underlying recognition engine without having to recompile.

Nuance parameters control:

- The trade-off between speed and accuracy
- How long a talker must pause before the system decides the speaking is finished
- What audio provider to use, what volume levels to set, and so forth
- How many ranked recognition results to return
- The machine and port number where the client can find the server
- Whether and where the recognition client should save waveform files

These parameters are only a sampling of those that are available to you and that are described in detail here. A complete list of Nuance parameters begins on page 238.

WAYS TO SET NUANCE PARAMETERS

You can modify a Nuance parameter in any of three ways:

- From the command line when launching a program
- By creating a *nuance-resources* file
- From within your application, via a Nuance System function call

These three methods are appropriate in different circumstances.

To begin, consider the following example, which illustrates two of the three ways in which you can modify the workings of the Nuance System—experimentally at first, and then locking in desired changes. Once a talker has begun to speak to the Nuance System, the system must decide when the talker has finished. It makes this decision based on how long the person is silent after speaking. The parameter `ep.EndSeconds` specifies how long this silence should be. For instance, if this parameter is set to equal 2 seconds, and the talker pauses for 1 second but then continues speaking, the Nuance System does not stop listening. When the talker is finally finished and stops speaking for 2 or more seconds, the Nuance System stops listening and completes the recognition process.

The default value of `ep.EndSeconds` is 0.75 second, meaning that if the talker stops for more than three quarters of a second, the Nuance System decides that the talker is finished. If you run an application as follows:

```
% Xapp -package mypackage
```

you might decide that 0.75 second is too short and would want to experiment with a longer value. To try a value of 1.5 seconds, you would simply run:

```
% Xapp -package mypackage ep.EndSeconds=1.5
```

After trying several values, you might decide to make 1.2 the permanent value for this particular application. This is a very simple change to make. To modify the behavior of this recognition package, you would edit the file *mypackage/nuance-resources* (which does not exist if you or the developer has not created it, and which should not be confused with *mypackage/nuance-resources.defaults*), and add the line:

```
ep.EndSeconds      1.2
```

The next time you run the application, the Nuance System will find this new value in the recognition package and use it instead of the default.

If you recompile the grammar, the file *mypackage/nuance-resources* will not be overwritten, so the modifications remain in place.

SETTING PARAMETERS FROM THE COMMAND LINE

Nuance parameters can be modified from the command line of a Nuance System program. Most programs accept Nuance parameters from the command line, including:

```
batchrec  
playwav  
recclient  
recserver  
resouce-manager  
sample-application  
Xapp
```

To set a Nuance parameter from the command line, specify the parameter, without a leading dash, followed by *=value*, for example:

```
% batchrec -package mypackage ep.EndSeconds=1.2 \  
rec.DoNBest=TRUE rec.RejectWeight=-30 \  
-testset mytestset
```

In this example, *batchrec* builds a *configuration* object based on the package *mypackage*. It then overrides certain values in that configuration, setting *ep.EndSeconds* to 1.2, *rec.DoNBest* to TRUE, and *rec.RejectWeight* to -30. *batchrec* then begins initialization using the configuration that includes the command-line values.

All Nuance parameters have an associated data type, which is either Boolean, integer, float, or character string. Many parameters also have ranges of valid values. When reading the command line, the Nuance System checks the type and range of all parameters specified there. If any values are of the wrong type or are out of range, it prints an error and exits. The only valid values for Boolean parameters are TRUE and FALSE. These values are case-sensitive, so capital letters should always be used.

- **Note:** Do not place string parameter values in quotation marks when setting them from the command line. You may, however, place the entire command-line argument in quotes. For example, do not do this:

```
% Xapp audio.Provider="cfone"
```

The following is alright, though:

```
% Xapp "audio.Provider=cfone"
```

Supporting Nuance Command-Line Parameters in Your Application

You can easily extend applications using the Nuance System to accept Nuance parameters on the command line, by using the Nuance function `NuanceConfigBuildFromCommandLine()` to process command-line arguments. See the `NuanceConfigBuildFromCommandLine()` online documentation for more information. The use of this function does not preclude your program from having its own command-line options; in fact, Nuance parameters can be interleaved with other arguments. Nuance will extract the arguments intended for it, leaving all others—for example,

```
% myapp -myarg1 -package mypackage -myarg2 myval2 \
    audio.Device=/dev/sound/2
```

In this example it is assumed that the application *myapp* calls `NuanceConfigBuildFromCommandLine()` to process its command-line arguments. The function `NuanceConfigBuildFromCommandLine()` will look at the command line, identify the arguments `-package mypackage` and `audio.Device=/dev/sound/2` as belonging to Nuance, use them to build its configuration object, and remove them from the command line. After `NuanceConfigBuildFromCommandLine()` returns, *argc* and *argv* will have been modified to look as if the original command line was

```
% myapp -myarg1 -myarg2 myval2
```

The application can then process these command-line arguments, which were intended for it.¹ Thus, you can write applications supporting all the Nuance

¹This means that you should always call `NuanceConfigBuildFromCommandLine` before processing your own command-line arguments.

System command-line options by adding a single function to your programs. Nuance recommends that you write all applications to include this flexibility.

Identifying Nuance Command-Line Parameters

As the function `NuanceConfigBuildFromCommandLine()` reads the command line, it must decide whether each argument belongs to the Nuance System or to your program. The logic behind this decision is covered in detail in the `NuanceConfigBuildFromCommandLine()` online documentation, but two main principles determine command-line ownership:

- The command-line switch `-package` and the single argument following it always belong to the Nuance System.
- Any argument of the form `string1.string2=string3` belongs to the Nuance System, unless it is preceded by the switch `-notnuance`.

SETTING PARAMETERS FROM A RESOURCE FILE

You may want to associate certain Nuance parameter settings with a specific recognition package. For instance, you might decide that a certain recognition system runs best with recognition pruning set to 900. Rather than always running applications with the command-line argument `rec.Pruning=900` when you use that recognition package, it would be cleaner to associate that modification with the recognition package itself.

To associate a Nuance parameter setting with a recognition package, simply create a file called *nuance-resources* in your recognition package directory (the directory created by the *nuance-compile* program). Add the following line to the newly created file:

```
rec.Pruning    900
```

or:

```
rec.Pruning=900
```

The next time any application points to this recognition package, the pruning value will be set to 900.

- **Note:** Each recognition package contains a file called *nuance-resources.defaults*. Do not modify this file, because it will be overwritten the next time you compile your recognition grammar. Instead, create a new file called *nuance-resources*. When you recompile your recognition grammar, the *nuance-resources* file will not be modified.
- **Note:** In the case where your grammar file is called *mypackage.grammar*, it may be more convenient to place your parameter changes in a file called *mypackage.nuance-resources*. Each time that you recompile your package, *nuance-compile* will copy *mypackage.nuance-resources* to *mypackage/nuance-resources*. This behavior is very convenient, since all of your source files (e.g., grammar, slot definitions, parameter settings) are in the same directory. If your compiled package is inadvertently removed, all the sources are available to regenerate it, including your parameter settings.
- **Note:** Your *nuance-resources* file overrides all the values in *nuance-resources.defaults*. You don't need to copy the *nuance-resources.defaults* file, since the Nuance System will read that file as well. Just use *nuance-resources* to specify the parameters that you want to change.

You can include multiple Nuance System parameters in a *nuance-resources* file, one per line. Begin the variable name in the first column, and separate the value from the name by spaces or tabs. Lines beginning with ";" are considered comments, and are ignored.

SETTING PARAMETERS FROM WITHIN AN APPLICATION

You will sometimes want to modify Nuance parameters from within a running program. For instance, `ep.EndSeconds=0.5` second might be appropriate when a yes/no question is asked. But `ep.EndSeconds=4.0` seconds would be more realistic when users are asked to record comments about their experiences with your application.

To change such values while the application is running, the Nuance System provides parameter-setting functions. In the RecClient API, these functions are:

```
NuanceStatus RCSetIntParameter (RecClient *rc,
                                char *param, int value);
```

```
NuanceStatus RCSetFloatParameter (RecClient *rc,
    char *param, float value);
```

```
NuanceStatus RCSetStringParameter (RecClient *rc,
    char *param, char *value);
```

The `RCSetIntParameter()` functions can also be used to set Boolean parameters; allowable values are 1 and 0.

In the `RCEngineInterface` class, implemented by `RCEngine` and `RCEngineRemote`, these functions are:

```
NuanceStatus SetParameter(char const * name,
    char const * value, unsigned id)
```

```
NuanceStatus SetParameter(char const * name, int value,
    unsigned id)
```

```
NuanceStatus SetParameter(char const * name, double value,
    unsigned id)
```

See the online documentation for these functions for more details.

LIMITATIONS ON SETTING PARAMETERS

Not all parameters can be set at all times. Some cannot be set at all, but can only be queried. Setting parameters from the command line or from a *nuance-resources* file is referred to as *setting at initialization*. Setting parameters using the *SetParameter* functions is called *setting at runtime*. Because some parameters affect the way in which the Nuance System initializes itself, those parameters can be set only at initialization. Others are more flexible and can also be set at runtime. Some parameters are set by the Nuance System as a function of the recognition process. Examples of such parameters are `rec.SamplingRate` and various statistics extracted from the recognition process. These parameters are not settable at all, but can only be queried.

Each of these parameters is explained on pages 238 through 252 and is marked in Table 14 as Init-settable, runtime-settable, neither, or both. If you attempt to set a non-Init-settable parameter at initialization, you get a warning message, and the `NuanceConfig` function returns a null `NuanceConfig` object, causing most applications to exit. If you attempt to set a non-runtime-settable parameter, the `SetParameter` function returns an error code.

TABLE 14 : PARAMETER DATATYPES AND VALUES

Nuance Parameter	Runtime-gettable	Init-settable	Runtime-settable	Type	Default	Min ^a	Max ^b
audio.BargelnSNR	No	Yes	No	INT	10	5	30
audio.Device	Yes	Yes	No	STRING	—	—	—
audio.Format	Yes	Yes	No	STRING	—	—	—
audio.InputSource	Yes	Yes	Yes	STRING	—	—	—
audio.InputVolume	Yes	Yes	Yes	INT	—	0	255
audio.OutputDest	Yes	Yes	Yes	STRING	—	—	—
audio.OutputVolume	Yes	Yes	Yes	INT	—	0	255
audio.Provider	Yes	Yes	No	STRING	—	—	—
audio.SupportsTelephony	Yes	No	No	BOOL	—	—	—
cfone.SerialPort	No	Yes	No	STRING	—	—	—
client.AllowBargeln	Yes	Yes	No	BOOL	FALSE	—	—
client.Behaviors	Yes	Yes	Yes	STRING	timeout	—	—
client.FilenameRecorded	Yes	No	No	STRING	—	—	—
client.FindOrCreateServer	No	Yes	No	STRING	find	—	—
client.KillPlaybackOnBargeln	Yes	Yes	Yes	BOOL	TRUE	—	—
client.MinPreSpeechSilenceSecs	Yes	Yes	Yes	FLOAT	0.025	0.01	20
client.NoSpeechTimeoutSecs	Yes	Yes	Yes	FLOAT	60	0.01	60
client.PromptIncrementalRead Ahead	No	Yes	No	BOOL	FALSE	—	—
client.PromptMaxReadAheadSecs	No	Yes	No	FLOAT	3	—	—
client.PromptMinReadAheadSecs	No	Yes	No	FLOAT	0.5	—	—
<p>a.Min = minimum value b.Max = maximum value c.At runtime, this parameter can be queried only through the NuanceConfig object with the function NuanceConfigGetStringParameter(), and cannot be set by the developer in any way. d.At runtime, this parameter can be queried only through the NuanceConfig object with the function NuanceConfigGetIntParameter(), and cannot be set by the developer in any way.</p>							

TABLE 14 (CONTINUED): PARAMETER DATATYPES AND VALUES

Nuance Parameter	Runtime- gettable	Init- settable	Runtime- settable	Type	Default	Min ^a	Max ^b
client.RecognizerTooSlowTimeout Secs	Yes	Yes	Yes	FLOAT	8	0	20
client.RecordCounter	Yes	Yes	Yes	INT	1	0	—
client.RecordDirectory	Yes	Yes	Yes	STRING	.	—	—
client.RecordFilename	Yes	Yes	Yes	STRING	utt%02d.wa v	—	—
client.ServerSetupTimeoutMs	No	Yes	No	INT	5000	1000	—
client.TooMuchSpeechTimeout Secs	Yes	Yes	Yes	FLOAT	30	0.1	60
client.WriteWaveforms	Yes	Yes	Yes	BOOL	FALSE	—	—
config.AckAbortTimeoutSeconds	No	Yes	No	FLOAT	20.0	1.0	—
config.DebugLevel	No	Yes	No	INT	—	0	4
config.GetSetTimeoutSeconds	No	Yes	No	FLOAT	10.0	1.0	—
config.InterpretationTimeout Seconds	No	Yes	No	FLOAT	5.0	1.0	—
config.LicenseCodesFile	No	Yes	No	STRING	\${NUANCE} /data/ license- codes	—	—
config.RCParentReadTimeoutMs	No	Yes	No	INT	10000	1000	—
config.RecClientHostname	No	Yes	No	STRING	localhost	—	—
config.RecClientPort	No	Yes	No	STRING	9200	—	—
config.RecClientTimeoutMsecs	No	Yes	No	INT	5000	5	30000
config.ServerDebugWindow	No	Yes	No	BOOL	FALSE	—	—

a.Min = minimum value

b.Max = maximum value

c.At runtime, this parameter can be queried only through the NuanceConfig object with the function NuanceConfigGetStringParameter(), and cannot be set by the developer in any way.

d.At runtime, this parameter can be queried only through the NuanceConfig object with the function NuanceConfigGetIntParameter(), and cannot be set by the developer in any way.

TABLE 14 (CONTINUED): PARAMETER DATATYPES AND VALUES

Nuance Parameter	Runtime-gettable	Init-settable	Runtime-settable	Type	Default	Min ^a	Max ^b
config.ServerExecutable	No	Yes	No	STRING	\${NUANCE}/scripts/recserver	—	—
config.ServerHostname	No	Yes	No	STRING	localhost	—	—
config.ServerPort	No	Yes	No	STRING	8200	—	—
config.ServerStderr	No	Yes	No	STRING	/dev/null	—	—
config.ServerStdout	No	Yes	No	STRING	/dev/null	—	—
config.SetWordGrammarTimeout Seconds	No	Yes	No	FLOAT	10.0	1.0	—
config.TcpSetupTimeoutMs	No	Yes	No	INT	5000	1000	—
config.VersionPackageTimeoutMs	No	Yes	No	INT	5000	1000	—
dtmf.Mask	Yes	Yes	Yes	STRING	none	—	—
ep.AdditionalEndSilence	Yes	Yes	Yes	FLOAT	0.65	0.00	5.0
ep.AdditionalStartSilence	Yes	Yes	Yes	FLOAT	0.35	0.00	5.0
ep.EndSeconds	Yes	Yes	Yes	FLOAT	0.75	0.20	10.0
ep.StartSeconds	Yes	Yes	Yes	FLOAT	0.30	0.06	1.0
ep.ThresholdSnr	Yes	Yes	Yes	INT	10	5	30
package.CreationDirectory	No	No	No	STRING	—	—	—
package.GrammarName% ^c	No	No	No	STRING	—	—	—
package.ID ^c	No	No	No	STRING	—	—	—
package.Location ^c	No	No	No	STRING	—	—	—
package.NLDefined ^d	No	No	No	BOOL	—	—	—
<p>a.Min = minimum value b.Max = maximum value c.At runtime, this parameter can be queried only through the NuanceConfig object with the function <code>NuanceConfigGetStringParameter()</code>, and cannot be set by the developer in any way. d.At runtime, this parameter can be queried only through the NuanceConfig object with the function <code>NuanceConfigGetIntParameter()</code>, and cannot be set by the developer in any way.</p>							

TABLE 14 (CONTINUED): PARAMETER DATATYPES AND VALUES

Nuance Parameter	Runtime-gettable	Init-settable	Runtime-settable	Type	Default	Min ^a	Max ^b
package.NumGrammars ^d	No	No	No	INT	—	—	—
prune.AdjustPruning	No	Yes	No	BOOL	TRUE	—	—
rec.BacktraceFinalsOnly	Yes	Yes	Yes	BOOL	TRUE	—	—
rec.ClashThreshold	Yes	Yes	Yes	INT	1000	—	—
rec.ClipConfidenceScore	No	Yes	No	Bool	TRUE	—	—
rec.ConfidenceRejection Threshold	Yes	Yes	Yes	INT	45	—	—
rec.ConsistencyThreshold	Yes	Yes	Yes	Int	3500	—	—
rec.DoNBest	No	Yes	No	BOOL	FALSE	—	—
rec.GenConfidence	Yes	Yes	Yes	BOOL	TRUE	—	—
rec.GenPartialResults	Yes	Yes	Yes	BOOL	FALSE	—	—
rec.GrammarWeight	No	Yes	No	FLOAT	5.0	0.0	100
rec.Interpret	Yes	Yes	Yes	BOOL	TRUE	—	—
rec.NumNBest	Yes	Yes	Yes	INT	10	1	500
rec.PartialResultSeconds	Yes	Yes	Yes	FLOAT	0.5	0.1	—
rec.pass1.gp.WTW	Yes	Yes	Yes	INT	—	-1000	1000
rec.PPR	Yes	Yes	Yes	BOOL	FALSE	—	—
rec.Pruning	Yes	Yes	Yes	INT	—	400	5000
rec.QuickNBest	No	Yes	No	BOOL	TRUE	—	—
rec.SamplingRate	Yes	No	No	INT	—	—	—
rec.SkipObsFrames	No	Yes	No	INT	0	0	1000000

a.Min = minimum value

b.Max = maximum value

c.At runtime, this parameter can be queried only through the NuanceConfig object with the function NuanceConfigGetStringParameter(), and cannot be set by the developer in any way.

d.At runtime, this parameter can be queried only through the NuanceConfig object with the function NuanceConfigGetIntParameter(), and cannot be set by the developer in any way.

TABLE 14 (CONTINUED): PARAMETER DATATYPES AND VALUES

Nuance Parameter	Runtime-gettable	Init-settable	Runtime-settable	Type	Default	Min ^a	Max ^b
rec.SuppressNullResult	No	Yes	No	Bool	TRUE	—	—
rm.Addresses	No	Yes	No	STRING	—	—	—
rm.ClientInhibitMs	No	Yes	No	INT	10000 0	—	—
rm.MaxClients	No	Yes	No	INT		128	1 920
rm.MaxPackages	No	Yes	No	INT		16	1
rm.MaxServers	No	Yes	No	INT	32	1	64
rm.Port	No	Yes	No	INT	7777	—	—
rm.TcpPort	No	Yes	No	INT	0	—	—
rm.UdpPort	No	Yes	No	INT	7777	—	—
server.MaxClients	No	Yes	No	INT	15	1	—
vrs.ConnectionType	No	Yes	No	STRING	utterance	—	—
vrs.FinalResultTimeoutMs	No	Yes	No	INT	30000	—	—
vrs.InitializationTimeoutMs	No	Yes	No	INT	30000	—	—
vrs.RecordDirectory	Yes	Yes	Yes	STRING	.	—	—
vrs.RecordFilename	Yes	Yes	Yes	STRING	utt%02d	—	—
vrs.ServerConnectionTimeoutMs	No	Yes	No	INT	10000	—	—
vrs.WriteEndpointed	Yes	Yes	Yes	BOOL	TRUE	—	—
vrs.WriteWaveforms	Yes	Yes	Yes	BOOL	FALSE	—	—
wavout.AutoFilenameExtension	Yes	Yes	Yes	BOOL	TRUE	—	—
wavout.FileFormat	Yes	Yes	Yes	STRING	sphere	—	—
<p>a.Min = minimum value b.Max = maximum value c.At runtime, this parameter can be queried only through the NuanceConfig object with the function NuanceConfigGetStringParameter(), and cannot be set by the developer in any way. d.At runtime, this parameter can be queried only through the NuanceConfig object with the function NuanceConfigGetIntParameter(), and cannot be set by the developer in any way.</p>							

QUERYING PARAMETERS

Although there are three ways to set Nuance parameters, you can only query them through the RecClient API or RCEngine *GetParameter* functions at runtime. The RecClient API provides the following functions:

```
NuanceStatus RCGetIntParameter(RecClient *rc,
                               char *param, int *value);

NuanceStatus RCGetFloatParameter(RecClient *rc,
                                  char *param, float *value);

NuanceStatus RCGetStringParameter(RecClient *rc,
                                   char *param, char *value, int maxlen);
```

The `RCGetIntParameter()` function can also be used to query Boolean parameters.

The `RCEngineInterface` class provides a single member function:

```
NuanceStatus GetParameter(char const * name, unsigned id)
```

This function generates a `ParameterGottenNotification` object that encapsulates the value of the parameter you specify by name.

See the online documentation for these functions for more details.

Restrictions

Some Nuance parameters cannot be queried at runtime, typically because the parameter is not owned by a single Nuance System module that can provide an authoritative value. Each parameter is marked in Table 14 as runtime-gettable or not. If the parameter in which you are interested is not runtime-gettable, it is typically a parameter designed for developer preference, and is either Init-settable or runtime-settable. Consider explicitly setting it so that you will know its value.

In Nuance version 6, RecClients are not continuously connected to RecServers. As such, certain parameters cannot be queried between sentences, since the RecClient is not connected to a RecServer that can provide an answer. In this case, the `GetParameter` function will return `NUANCE_NOT_CONNECTED`.

PARAMETER DEFINITIONS

The Nuance System currently has more than 50 parameters, and more are added with each new system version. Table 14 on page 232 shows each parameter's data type, default value, minimum and maximum values if any, and settable/gettable flags.

The format for naming Nuance parameters is *modulename.ParameterName*, where *modulename* indicates the Nuance System software module that owns the parameter.² Nuance parameters whose *modulename* is *config* or *package* are not owned by one module, but typically are referenced by many different modules at runtime. The *config* parameters are settable only at initialization and are not runtime-gettable. The *package* parameters are set by the Nuance System as a function of the recognition package that was loaded and can be queried only with the functions `NuanceConfigGetIntParameter()` and `NuanceConfigGetStringParameter()`.

Recognition Parameters

`rec.Pruning`

Controls the speed/accuracy trade-off by varying the width of the Viterbi beam search. A smaller number increases recognition speed, but decreases accuracy. See Chapter 2 for more detail.

`rec.SkipObsFrames`

A second control for the speed/accuracy trade-off. Recognition hypotheses that fall below this threshold are computed in a less expensive manner. A value of zero indicates that all hypotheses should be evaluated with the less expensive method. A value of 1000000 means that no recognition hypotheses should use this method. Zero and 1000000 are the only meaningful values.

²The *owner* of a parameter is the software module to which a *SetParameter* request for that parameter is routed.

`rec.GrammarWeight`

Controls the grammar processing weight—the relative weighting of acoustic and linguistic scores during recognition. See Chapter 2 for more detail.

`rec.pass1.gp.WTW`

Controls the word transition weight—the trade-off between inserted/deleted words. Most developers never modify this parameter.

`rec.SamplingRate`

Can be queried to reveal the audio sampling rate required by the acoustic models. This cannot be modified.

`rec.GenPartialResults`

Set this Boolean parameter to TRUE if you want the recognition engine to generate partial (intermediate) recognition results periodically. Each partial result will generate a `NUANCE_EVENT_PARTIAL_RESULT`. The frequency with which partial results are generated depends on `rec.PartialResultSeconds`.

`rec.ConsistencyThreshold`

Used during enrollment of pronunciations for new phrases to specify how similar an utterance needs to be to a previously enrolled pronunciation to be consistent (see Chapter 16). The smaller the consistency threshold, the closer the match between utterance and previous pronunciation needs to be for the utterance to be considered consistent. The default value is 3500.

`rec.ClashThreshold`

Used during enrollment of pronunciations for new phrases to specify how close two utterances must be to be considered clashing pronunciations (see Chapter 16). The default value is 1000. A smaller threshold reduces the number of clashes detected.

`server.MaxClients`

Set this parameter to determine the maximum number of clients that the server will service at a single instant. This number can change depending on what grade of service you would like per client connection. The default was arbitrarily chosen. As of version 6.0, it is not possible to service more than two stock-quote clients at a single time.

`prune.AdjustPruning`

Activates or deactivates an algorithm that detects *confusion* on the part of the recognizer. Nuance Communications recommends that you use this in its default (TRUE) setting. If you are experimenting with a recognition task that requires a large amount of computation (e.g., if *batchrec* often prints a PFSG_REALS value of greater than 750), you should experiment with setting this value to FALSE.

`rec.PartialResultSeconds`

Controls the frequency at which partial results, which are the recognizer's current best guess based on the data processed so far, should be computed. Partial results will be generated only if you register a callback for NUANCE_EVENT_PARTIAL_RESULT. See the `RCRegisterCallback()` and `RSRegisterCallback()` online documentation.

`rec.BacktraceFinalsOnly`

Indicates whether partial parses of the grammar are acceptable. When set to TRUE, the system returns a recognition result only if it represents a complete path through the recognition grammar. For example, if a recognition grammar requires 16 digits, but the talker speaks only 10 digits, the utterance is rejected if `rec.BacktraceFinalsOnly` is TRUE. Otherwise, the 10-digit response is returned.

`rec.DoNBest`

Indicates whether to return more than one recognition result in the `RecResult` structure. N-Best processing requires more memory than the standard 1-best method, but allows the application to rescore the set of results by using additional knowledge sources.

`rec.NumNBest`

Indicates the maximum number of recognition results to return under N-Best processing. It has no effect when `rec.DoNBest=FALSE`.

`rec.Interpret`

Indicates whether the recognizer should interpret the recognized string using natural language and add the `NLResult` object to the `RecResult` object.

`rec.PPR`

Specifies whether the recognizer should perform phonetic pruning. See Chapter 2 for more details.

Audio Prompt Parameters

With Nuance version 6, you can read audio prompt files incrementally. In essence, audio files are now streamed by the RecClient. The major advantage of this approach is that it reduces the time needed to start playing a prompt file. This is especially advantageous for large prompt files that may take a long time to load if done in their entirety. The following parameters control this behavior:

`client.PromptIncrementalReadAhead`

Controls whether the file is to be read incrementally or not. If set to `TRUE`, the RecClient reads prompt files incrementally. If set to `FALSE` (the default), it reverts to the old behavior, and reads the entire file before playing it.

The audio buffer for incremental reads has two triggers—a low-water (or sample) mark, and a high-water mark. When the number of samples in the buffer falls below the low-water mark, a read is triggered. Then, data are read in until the high-water mark is reached. Now, the data are played out, and the next read will be triggered by the data falling below the low-water mark.

`client.PromptMinReadAheadSecs`

Determines the low-water mark in seconds. It is a float parameter, so setting it to 0.5 (the default) implies that there should be at least half a second of data in the buffer. Increase this parameter if the prompt output seems to be broken up.

`client.PromptMaxReadAheadSecs`

Sets the high-water mark in seconds. It is a float parameter, and is set to 3.0 by default. This controls the amount of data to read up to. Setting it to a large value will mimic the older nonincremental reading behavior.

Recognition Client Parameters

`config.RecClientHostname`

Specifies the name of the host machine containing the RecClient that an RCEngine should register with.

`config.RecClientPort`

Specifies the port on the host machine specified in `config.RecClientHostname` providing the RecClient that an RCEngine should register with.

`config.RecClientTimeoutMsecs`

Specifies the number of milliseconds that an RCEngine has to find the specified RecClient before a timeout occurs.

`client.WriteWaveforms`

Indicates whether or not to save recognized utterances to disk.

`client.RecordDirectory`

Indicates the directory where the RecClient should save recognized waveforms. It has no effect if `client.WriteWaveforms=FALSE`.

`client.RecordFilename`

Indicates the filename where recognized utterances should be saved. If the first character is '/' the filename is considered to contain an absolute path. Otherwise, the value of `client.RecordDirectory`, plus a '/' character, is prepended to this name to form a complete filename. If the value includes a form of %d, `sprintf()` puts a counter that increments each utterance into the string, resulting in a filename that increments each utterance. It has no effect if `client.WriteWaveforms=FALSE`. For example, if the value of `client.RecordFilename` is `utt%03d.wav`, consecutive files will be saved as `utt001.wav`, `utt002.wav`, `utt003.wav`, and so forth. This counter can be set or queried via `client.RecordCounter`.

`client.RecordCounter`

The parameter `client.RecordFilename` supports an embedded "%d", which will be replaced by an increasing counter with each utterance recorded to disk. The parameter `client.RecordCounter` allows

applications to both query and modify the value of this RecClient internal counter.

`rec.GenConfidence`

Specifies whether the recognizer should generate a confidence score for each recognition result. See the discussion on confidence scoring on page 337 for more details.

`rec.ConfidenceRejectionThreshold`

When the confidence score for the recognition result is lower than the threshold, the system replaces the recognized hypothesis with an empty string. You can set this parameter to 0 to avoid any rejection based on recognition result's confidence scores.

`rec.ClipConfidenceScore`

Specifies whether to ensure that confidence scores in an N-Best list are monotonically decreasing.

`rec.SuppressNullResult`

When set to TRUE, the system removes recognition results in the N-Best list that have become empty strings because of having confidence scores lower than the threshold.

`client.AllowBargeIn`

When set to TRUE (default is FALSE), the system allows the caller to barge in on top of prompts. The current audio provider must provide the correct support for this to be available. For more information, see the short topic on barge-in on page 331.

`client.KillPlaybackOnBargeIn`

When the caller barges in on the prompt, the prompt immediately stops playing (as soon as speech is detected), if this parameter is set to TRUE, which is the default value.

`client.FilenameRecorded`

Can be queried to reveal the complete filename that was assembled from `client.RecordDirectory`, `client.RecordFilename`, and the incrementing counter, if used.

`client.ServerSetupTimeoutMs`

Specifies the maximum number of milliseconds to wait for a created `RecServer` to connect to the `RecClient` that created it.

`dtmf.Mask`

Specifies the set of touch tones to exclude from generating `NUANCE_EVENT_DTMF` events. To allow all touch tones, set this parameter to "none". To allow only digit touch tones, set it to "*#".

`client.Behaviors`

Specifies the set of behaviors to load into the `RecClient` at startup. Currently, the only supported value is *timeout*, which is the default. The *timeout* behavior installs timeout handlers for speech-too-early, no-speech, too-much-speech, and recognizer-too-slow timeouts. To disable the timeout behavior, set this parameter to the empty string—that is, `client.Behaviors=""`.

`client.MinPreSpeechSilenceSecs`

Determines the minimum number of seconds the user should be silent after `RCRecognize()` is called or after playback completes (in the case of barge-in).

This value is used only by the timeout behavior described earlier. If the user speaks before the required minimum number of prespeech silence, the timeout behavior (if active) will generate a fake final result with the result string *<speech_too_early>*. However, this fake result will be delayed until end-of-speech occurs.

Set this parameter to 0 to allow the user to speak at any time.

If the timeout behavior has not been loaded, this parameter is ignored.

`client.NoSpeechTimeoutSecs`

Determines the maximum number of seconds of silence to allow after `RCRecognize()` is called or after playback completes (in the case of barge-in).

This value is used only by the timeout behavior described earlier. If the user does not speak within this number of seconds, the timeout behavior (if active) will generate a fake final result with the result string *<timeout>*.

This parameter is set internally by the RecClient based on the timeout value passed into `RCRecognize()`. This means that setting the parameter via `RCSetFloatParameter()` is useless. It is currently for internal use only.

`client.TooMuchSpeechTimeoutSecs`

Determines the maximum number of seconds of speech to allow as soon as the user starts speaking.

This value is used only by the timeout behavior described earlier. If end-of-speech does not occur within this number of seconds, the timeout behavior (if active) will generate a fake final result with the result string `<too_much_speech_timeout>`.

Set this parameter to a maximum value of 60 to avoid this timeout from occurring (a minute is considered unacceptably long by Nuance standards).

If the timeout behavior has not been loaded, this parameter is ignored.

`client.RecognizerTooSlowTimeoutSecs`

Specifies the maximum number of seconds to wait for the final recognition result from the recognizer after the user has finished speaking.

This value is used only by the timeout behavior described earlier. If a final result has not arrived within the specified number of seconds after end-of-speech was detected, the timeout behavior generates a fake final result with the result string `<recognizer_too_slow>`.

Set this parameter to a maximum value of 20 to avoid the timeout from happening as much as possible.

If the timeout behavior has not been loaded, this parameter is ignored.

Networking Parameters

`client.FindOrCreateServer`

Indicates whether the RecClient should find an existing *recserver* or use the system call `fork()` to create one. The default value is `find`, indicating that the RecClient should find a running *recserver* on the machine and port specified by `config.ServerHostname` and `config.ServerPort`, and

connect to that. Other allowable values are *find-or-create*, *create*, and *neither*.

`config.ServerHostname`

Specifies the machine name or IP address where the RecClient should look to find a *recserver* running.

`config.ServerPort`

Specifies the port number where the *recserver* should listen for connections, and the port to which the RecClient should attempt to connect to find a *recserver*.

Audio Parameters

`audio.Provider`

Selects the audio/telephony device to use for live audio playback, recording, and call control. See Chapter 15 for more information.

`audio.Device`

Specifies the name of the audio device that should be opened for input and output. The meaning of this value depends on the audio provider selected. For example, the *dialogic* provider defines this parameter to indicate a channel number, whereas the *native* provider uses it to refer to a UNIX audio device, such as `/dev/sound/1` on a Sun workstation. This parameter is optional.

`audio.Format`

Sets the audio sampling rate, audio precision, and audio encoding. The syntax of this parameter is somewhat flexible. You can specify any subset of the three components, in any order. For identification, the components must be separated by hyphens. The sampling rate must be specified as a number of thousand samples, followed by the letter 'k' (upper- or lowercase). The precision must be a number followed by the string "bit", "Bit", or "BIT". The encoding must be one of the following lowercase strings: "linear," "mulaw," or "alaw." In general, you should try to avoid being too specific when specifying this parameter, because most audio providers support only a limited number of formats. For example, most telephone-based audio providers support only "8k-8bit-mulaw" or "8k-

8bit-alaw." In addition, the RecClient initialization process must try to match the specified rate to that which the *recserver* requires for the model set it is using during recognition. In essence, the RecClient tries to generate the intersection of the three formats specified by this parameter, the *recserver*, and the selected audio provider. Some syntactically valid examples for this parameter are 8bit, 16bit-linear, linear-16k-16BIT, and 8K-mulaw. This parameter is optional.

`audio.InputVolume`

Sets the audio input volume on a machine-independent scale of 0 to 255. Be aware that the scale is not guaranteed to be linear. The actual measured volume level for a given value is unlikely to be the same on different platforms and for different audio providers.

`audio.OutputVolume`

Sets the audio output volume on a machine-independent scale of 0 to 255.

`audio.InputSource`

Indicates whether audio should be sampled from the microphone port or the line-in port. Allowable values are *mic*, *line*, and *digital*. This parameter is relevant only for the *native* and *cfone* audio providers.

`audio.OutputDest`

Indicates the output port to which playback should be directed. Allowable values are *speaker*, *line*, and *headphone*. This parameter is ignored on Silicon Graphics machines because audio output destination is not software controllable. On Silicon Graphics machines, plugging an 1/8th-inch connector into the headphone port automatically silences output from the speaker. This parameter applies only to the *native* and *cfone* audio providers.

`audio.BargeInSNR`

See the short topic "Barge-In" on page 331 for details about this parameter.

`audio.SupportsTelephony`

Gettable only—allows an application to determine whether the audio provider selected is telephone-based.

- **Note:** Audio-provider-specific parameters generally take the form *audio.providername.ParameterName*. See Chapter 15 for a list and description of these parameters.

`cfone.SerialPort`

Specifies the UNIX serial device name the Computerfone is connected to. This parameter is optional.

Endpointing Parameters

`ep.StartSeconds`

Controls the minimum duration of high-energy speech required to indicate that the talker has begun to speak. Most developers never modify this parameter.

`ep.EndSeconds`

Controls the minimum amount of silence required to indicate that the talker has finished speaking. A longer value is less likely to cut the talker off, but also makes the interaction less snappy. This parameter is often set from within an application, depending on the type of question being asked. For instance, 0.5 second might be appropriate for a simple yes/no question, whereas 1.5 seconds might be the right value when a talker is likely to pause mid-sentence.

`ep.AdditionalStartSilence`

`ep.AdditionalEndSilence`

`ep.ThresholdSNR`

Provide additional endpointing control, which most applications should not have to change. For more details, see the short topic on endpointing on page 329.

Package Parameters

The following parameters are associated with the recognition package. Unlike other Nuance parameters, these parameters exist only in the `NuanceConfig` object. They can only be read using the function `NuanceConfigGetStringParameter()` or the function

`NuanceConfigGetIntParameter()`, and they cannot be set using any API function.

`package.Location`

Indicates the name of the directory from which the package was loaded.

`package.CreationDirectory`

Indicates the name of the directory where the package was initially compiled. This is only for information. The application should not look for any files in that directory, since the package may have been moved.

`package.ID`

A character string that uniquely identifies the grammar, acoustic models, and software version used to compile the package. Can be used to verify that two programs or two `NuanceConfig` objects are using the same recognition package.

`package.NumGrammars`

Indicates the number of top-level recognition grammars compiled into the package.

`package.GrammarName%d`

If there are N top-level recognition grammars (as indicated by `package.NumGrammars`), their names can be read from the `NuanceConfig` object as `package.GrammarName0` through `package.GrammarName $N-1$` .

`package.NLDefined`

A Boolean variable indicating whether or not natural language understanding commands were compiled into this package.

Configuration Parameters

`config.LicenseCodesFile`

Specifies a file other than the standard `$NUANCE/data/license-codes` where the Nuance license for this machine may be found.

config.ServerExecutable

Specifies an executable other than *\$NUANCE/scripts/recserver* the RecClient should run in the background if it cannot find a *recserver* running on the machine specified by `config.ServerHostname` at the port specified by `config.ServerPort`.

config.ServerDebugWindow

Indicates with a TRUE value that if the RecClient creates a *recserver* in the background, it should appear in an *xterm* so that its output can be seen. You must have your DISPLAY environment variable properly set and *xterm* must be present in one of the directories in your path for this option to be effective when set to TRUE.

config.ServerStdout

Indicates the filename where the stdout of a *recserver* created by a RecClient should be written. This parameter is ignored if `config.ServerDebugWindow` is TRUE.

config.ServerStderr

Indicates the filename where the stderr of a *recserver* created by a RecClient should be written. This parameter is ignored if `config.ServerDebugWindow` is TRUE.

config.DebugLevel

Read by the RecClient, the RecClient API, and eventually other Nuance programs to determine how much debugging information to print to *stderr* according to the values in Table 15. Most programs default to a value of 2.

- Note: Nuance Communications does not recommend a value of less than 2 because it would hide important warnings.

TABLE 15: DEBUGGING PRINT VALUES

Value	Fatal Errors	Nonfatal Errors	Warnings	Information	Counters
0	x				
1	x	x			
2	x	x	x		

TABLE 15: DEBUGGING PRINT VALUES

Value	Fatal Errors	Nonfatal Errors	Warnings	Information	Counters
3	x	x	x	x	
4	x	x	x	x	x

`config.TcpSetupTimeoutMs`

Specifies the maximum time (in milliseconds) allowed for the TCP/IP connection associated with data transfer to be established. The default value should be adequate unless the client and server are (topologically) very far apart, or one of the machines is extremely busy.

`config.GetSetTimeoutSeconds`

Specifies the maximum time (in seconds) allowed for any Nuance subsystem to set or retrieve a Nuance parameter. The default should be sufficient.

`config.VersionPackageTimeoutMs`

Specifies the maximum time (in milliseconds) allowed for the Nuance System to verify that the client and server are using the same version of the software. The default should be sufficient.

`config.AckAbortTimeoutSeconds`

Specifies the maximum time (in seconds) allowed for an abort request to complete. Failure to complete within this time limit is a serious error. If increasing this value does not solve the problem, contact Nuance technical support.

`config.RCParentReadTimeoutMs`

Specifies the maximum time (in milliseconds) allowed in the RecClient API programs before a conclusion that a communication problem has developed between the parent and child processes.

`config.SetWordGrammarTimeoutSeconds`

Specifies the maximum time (in seconds) the RecClient object should wait for the server to set a dynamic grammar at a specific label. See the online documentation on `RCSetWordGrammar()` for more information.

`config.InterpretationTimeoutSeconds`

Specifies the maximum time (in seconds) the RecClient object should wait for the server to interpret a phrase using the NL engine. See the online documentation on `RCInterpret()` for more information.

- **Note:** Developers who run Purify or other memory-checking software may need to increase all of these time-out parameters. Doubling or even quadrupling the default values would be appropriate, because memory-checking software dramatically slows the speed of a program's execution.

Utterance Parameters

`wavout.FileFormat`

Selects the output format of the utterance file saved to disk by the RecClient. The following formats are currently supported:

- `sphere`—NIST format, extension `.wav` (this is the default)
- `def1`—Periphonics intermediate format; useful on the Periphonics platform when the utterance needs to be played back as a prompt at a later time, extension `.def1`
- `au`—Sun audio format, extension `.au`

`wavout.AutoFilenameExtension`

Determines whether the RecClient should attempt to determine the utterance filename extension automatically based on the format.

For example, if `wavout.FileFormat=au` and `client.RecordFilename=foo`, then the actual filename recorded (which can be queried by getting the Nuance string parameter `client.FilenameRecorded` at some point after end-of-speech has occurred) will be `foo.au`.

The default value for this parameter is `TRUE`, which means the RecClient looks at the format in which the file should be saved, determines whether the filename specified by the parameter `client.RecordFilename` already has an extension of that type, and, if it does, leaves it alone. Otherwise it appends the extension to it before saving.

By setting this parameter to `FALSE`, you tell the `RecClient` to save the utterance with the filename as is. However, if the `RecClient` fails to write out the filename (e.g., because the disk is full), it may still write out the file `tmp.<ext>`, where `<ext>` is the extension matching the file format.

VRSAPI Parameters

`vrs.ConnectionType`

Selects the connection type between the VRSAPI and the recserver when resource managers are being used. One or more resource managers will be used when the parameter `rm.Addresses` is set. The parameter `vrs.ConnectionType` has no effect when `rm.Addresses` is not set (the default), in which case the VRSAPI connects to the recserver specified by the parameters `config.ServerHostname` and `config.ServerPort` continuously.

The parameter `vrs.ConnectionType` has three valid values: `utterance`, `call`, and `continous`:

- When it is set to `utterance`, the VRSAPI connects to a recserver every time a new utterance is started. It disconnects when that utterance has been recognized or aborted.
- When it is set to `call`, the VRSAPI stays connected to the recserver for the duration of a phone call or until the audio channel is reset, whichever happens first.
- When it is set to `continous`, the VRSAPI connects to the recserver identified by the parameters `config.ServerHostname` and `config.ServerPort`, and stays connected to that server throughout. In this mode, the VRSAPI does not connect to a resource manager.

`vrs.InitializationTimeoutMs`

Specifies the maximum number of milliseconds to allow an instance of the VRSAPI to initialize after being constructed. If this timeout expires, an initialization-completed message is generated with a timeout status, which typically causes the application to fail at startup.

`vrs.ServerConnectionTimeoutMs`

Determines the maximum number of milliseconds to allow the VRSAPI to connect to a recserver. The default value should be adequate unless the client and server are (topologically) very far apart or one of the machines is extremely busy. If this timeout expires, an exception is thrown, which typically causes the current utterance to be rejected.

`vrs.FinalResultTimeoutMs`

Specifies the maximum number of milliseconds the VRSAPI should wait for the server to return the final recognition result after the utterance has ended. The default value should be adequate unless the server is extremely busy. This parameter is meant as a safety mechanism. Typically, the application (or application tools such as the dialog builder) will have its own timeout. If this timeout expires, the utterance is rejected.

`vrs.WriteWaveforms`

Specifies whether recognized utterances should be saved to disk. The default value is FALSE.

`vrs.WriteEndpointed`

If `vrs.WriteWaveforms` is set to TRUE, this parameter specifies whether the saved utterances are endpointed (TRUE) or saved as raw audio data (FALSE).

`vrs.RecordDirectory`

If `vrs.WriteWaveforms` is set to TRUE, this parameter specifies the directory the waveforms are saved in.

`vrs.RecordFilename`

If `vrs.WriteWaveforms` is set to TRUE, this parameter specifies the naming convention for saved waveforms. You can include the %d conversion character in the name if you want to designate incremental filenames for each saved utterance. For example, the default name is "utt%02d" and creates the file names `utt01.wav`, `utt02.wav`, and so on.

Resource Manager Parameters

`rm.Port`

When applied to a Resource Manager, denotes the port that the Resource Manager will employ for communications. When applied to a RecServer or RecClient, denotes the default port used in parsing the `rm.Addresses` parameter.

`rm.Addresses`

Indicates the location (hostname and port) of the Resource Manager (or Managers) with which a RecServer or RecClient should communicate. The general format is:

```
<hostname>[:<port>][,<hostname>[:<port>]]*
```

which is a comma-separated list of machine names, with optional port values. If the port is not specified, the value of `rm.Port` is assumed. Thus, the parameter specification:

```
rm.Port=1234 rm.Addresses=machine1,machine2
```

is precisely equivalent to:

```
rm.Addresses=machine1:1234,machine2:1234
```

This parameter is ignored when applied to a Resource Manager.

`rm.MaxServers`

Specifies the maximum number of RecServers that may connect to a given Resource Manager. This parameter is ignored if applied to anything other than a Resource Manager.

`rm.MaxClients`

Specifies the maximum number of RecClients that may connect to a given Resource Manager. This parameter is ignored if applied to anything other than a Resource Manager.

`rm.MaxPackages`

Specifies the maximum number of unique package names that may be managed by a given Resource Manager at any one time. Only packages that are either required by a connected RecClient or supported by a

connected RecServer are counted. This parameter is ignored if applied to anything other than a Resource Manager.

`rm.ClientInhibitMs`

Specifies the number of milliseconds for which the Resource Manager will give preferential service to RecServers that want to connect. This gives the Resource Manager time to determine the capabilities of the RecServers before any client requests arrive. It is unlikely that you will need to alter this parameter. This parameter is ignored if applied to anything other than a Resource Manager.

SHARP DECLARATION
EXHIBIT A
(Part 2 of 2)

15

NUANCE AUDIO AND TELEPHONY INTERFACES

The primary purpose of the Nuance RecClient is to perform audio playback and recording, and usually to control an associated telephone connection.

The RecClient supports a variety of audio/telephony interface devices. In Nuance terminology, each audio/telephony interface is called an *audio provider*.

There are two types of Nuance audio providers: those that provide audio via a telephone-based interface (such as a Dialogic telephone line card) and those that do not (such as the microphone and speaker on a Sun or NT workstation). Table 16 lists Nuance audio providers.

TABLE 16: AUDIO PROVIDERS

Audio Provider Name	Description	Supports Telephony	Additional Hardware or Software Required
native	Desktop audio on Sun workstations and PCs running Solaris/x86	NO	Sun: NO Intel: YES
cfone	Computerfone analog telephone interface	YES	YES
dialogic	Telephone line interface, optionally includes Antares card	YES	YES

AUDIO PROVIDER PLATFORMS

Table 17 lists the combinations of hardware platform and audio provider supported by the Nuance System.

TABLE 17: HARDWARE PLATFORM AND AUDIO PROVIDER COMBINATIONS

\$MACHINE_TYPE^a	Audio Provider Name		
	native	cfone	dialogic
sparc-solaris	X ^b	X	
i386-solaris	X ^b		X ^c
sco			X ^b
rs6000			X ^b
win32	X ^b		X ^c

- a. For information on the supported platforms and the MACHINE_TYPE environment variable, see the Nuance *Getting Started* manual.
- b. This is the default provider for the platform.
- c. If no SoundBlaster is present, Dialogic becomes the default.

SELECTING AN AUDIO PROVIDER

The Nuance audio provider frameworks allow you to write applications that are independent of the physical audio/telephony interface. An audio provider other than the default can be selected at runtime through the Nuance configuration parameter `audio.Provider`. If you run:

```
% sample-application -package my_package
```

on a Sun workstation, *sample-application* will read audio from the desktop microphone and play output through the built-in speaker. If you rerun *sample-application*, adding the argument `audio.Provider=cfone` as follows:

```
% sample-application -package my_package
audio.Provider=cfone
```

sample-application will instead wait for a phone call on the Computerfone, answer it, and thereafter read audio input from the telephone line.

You can easily write your applications to have this same flexibility, as described in the *Staying Flexible* discussion on page 278.

In general, only the `audio.Provider` parameter must be set to select the audio provider of interest. However, some audio providers may require you to specify additional Nuance configuration parameters to successfully connect to the desired device.

Some Nuance parameters are common among all audio providers and allow the application to control attributes such as the input and output volumes, physical device name, and audio format the provider should use. All of these standard parameters are described in detail in the discussion of configuration parameters in Chapter 14. Nuance audio provider configuration parameters start with the `audio` prefix. Parameters specific to each audio provider are *not* covered in Chapter 14. Instead, they are included here in the description of each audio provider.

AUDIO PROVIDER RESOURCE LOCKING

To avoid contention between two competing processes for the same channel of a given audio provider on a machine, Nuance audio providers use a file-based locking scheme. If the RCAPI initialization fails with a return status of `NUANCE_TELEPHONY_DEVICE_BUSY` or `NUANCE_AUDIO_DEVICE_BUSY`, some other process is already using the requested audio/telephony channel.

To find out which process is currently using the resource, use the *resource-locks* tool, described on page 319.

AUDIO PROVIDER TELEPHONY SUPPORT

Nuance audio providers can be grouped according to whether or not they support telephony actions such as answering the phone, detecting hangup, and transferring a call.

The majority of Nuance audio providers are associated with telephone interface hardware, and therefore support telephony. These providers require the application to be *telephony savvy*, and to perform call control in addition to playback and recording/recognition.

The only Nuance audio provider that does not support telephony is the *native* provider. The native provider can play, record, and recognize using RCAP functions. It does not, however, support control of any telephony interface, which means that audio data are available as soon as the RecClient completes its background initialization. The channel stays open until the RecClient object is terminated.

The native provider is mainly used for debugging, but can also be incorporated into desktop speech recognition applications.

An application can determine whether the runtime-selected audio provider supports telephony by querying the Nuance parameter `audio.SupportsTelephony`.

AUDIO PROVIDERS

Dialogic Audio Provider

Dialogic Corporation designs and builds telephone interface boards for PC-compatible computers. Nuance software supports recording, playback, and DTMF detection for all supported Dialogic cards. When the configuration includes Dialogic's Antares card, barge-in and endpointing are also provided.

Supported Dialogic Hardware Configurations

Digital and analog telephony support for Dialogic cards is detailed in Tables 18 and 19.

TABLE 18: SUPPORTED DIGITAL DIALOGIC LINE CARDS

Card Set	Bus	Channel	Notes
D/240SC-T1	SCBUS/PEB	12/24	12 full-duplex, 24 half-duplex
D/240SC-T1 with D/240SC expansion	SCBUS/PEB	24	24 full-duplex
DTI/240SC with Antares 2000/50	SCBUS/PEB	24	24 full-duplex, echo cancellation and endpointing on Antares
Other combinations of T1 or E1 network interfaces and DSP voice-resources/ Antares may also work.			

TABLE 18: SUPPORTED DIGITAL DIALOGIC LINE CARDS

Card Set	Bus	Channel	Notes
D/300SC-E1	SCBUS/PEB	16	16 full-duplex
DT1/300SC-E1 with Antares 2000/50	SCBUS/PEB	24	24 full-duplex, echo cancellation and endpointing on Antares
Other combinations of T1 or E1 network interfaces and DSP voice-resources/ Antares may also work.			

TABLE 19: SUPPORTED ANALOG DIALOGIC LINE CARDS

Card Set	Bus	Channel	Notes
D/41ESC	SCBUS/PEB	2/4	2 full-duplex, 4 half-duplex
D/121B with LSI-120	PEB	6/12	6 full-duplex, 12 half-duplex
D/41ESC with Antares 2000/50	SCBUS/PEB	4	4 full-duplex, echo cancellation and endpointing on Antares— one Antares supports 4 D/41ESC cards for 16 channels
Other combinations may be possible.			

External telephony support, detailed in Table 20, can be provided by an Antares 2000/50 and any of the network interface cards listed in Tables 18 and 19. The assumption is that the telephony card itself is being controlled by some third-party software, and the Antares is being used by Nuance software to perform echo cancellation and endpointing.

TABLE 20: EXTERNAL TELEPHONY SUPPORT

Card Set	Bus	Channel	Notes
Antares 2000/50 with D/41ESC, D/121B, D/240SC-T1, DTI/240SC	SCBUS/PEB	16	16 full-duplex channels, echo cancellation and endpointing on Antares.

Each instance of the *dialogic* audio provider attaches to one of the telephone channels supplied by a Dialogic board. Each channel is identified by a number between 1 and the number of available channels. For example, `audio.Device=3` selects the third telephone channel on the Dialogic board.

By default, full-duplex mode is enabled, requiring twice as many DSP resources as with half-duplex. To double the number of available half-duplex lines, set the `audio.dialogic.FullDuplex` parameter to `FALSE`. The disadvantage of this is that you cannot play and record at the same time (which is a requirement for barge-in).

Pertinent information about configuration parameters is in Tables 21 and 22. All `audio.dialogic` parameters in Table 22 are passed through the `RCSetStringParameter` and `RCGetStringParameter` calls; therefore, the data type is always a character string. Numeric values must be converted to ASCII string before being passed

RecClient Support for Dialogic's Antares Card

Nuance Version 6 supports Dialogic's Antares DSP card. With this release, the Nuance System supports 16 to 24 ports of playback, recording, DTMF detection, barge-in, and endpointing on a single Antares card.

TABLE 21: NOTES ON STANDARD NUANCE CONFIGURATION PARAMETERS FOR THE *dialogic* AUDIO PROVIDER

Parameter	Operation
<code>audio.Provider</code>	Can be set to <code>dialogic</code> to explicitly select this provider
<code>audio.Device</code>	The Dialogic channel/line for listening (default=0 → select next available one)
<code>audio.InputSource</code>	Irrelevant
<code>audio.OutputDest</code>	Irrelevant
<code>audio.InputVolume</code>	Operational when using Antares (default=128) (range=0–255)
<code>audio.OutputVolume</code>	(default=128) (range=0–255)

**TABLE 22 : ADDITIONAL NUANCE CONFIGURATION PARAMETERS
FOR THE *dialogic* AUDIO PROVIDER**

Parameter	Operation
General Dialogic Telephony Parameters	
audio.dialogic.Network	Selects either the T1 or analog configuration (a default is determined automatically)
audio.dialogic.FlashMsecs	Sets the number of milliseconds to put the line on hook for a flash-hook operation (default=600)
audio.dialogic.PauseMsecs	Sets the number of milliseconds a ',' in a dial string pauses (default=1000)
audio.dialogic.NumDIDDigits	The number of digits to expect for DID (default=7)
audio.dialogic.Transfer String	Sets the dialing format to be used to perform a call transfer when using hook-flash. Defaults to &, %s, & on analog lines and &%s on T1 lines. The %s in the format is replaced by the number to be dialed.
audio.dialogic.Lines	Used with DID to setup the range of lines which should be monitored for incoming calls.
GlobalCall Specific Parameters	
audio.dialogic.GlobalCall	Set to specify that Dialogic GlobalCall is to be used for telephony. This option may not be supported on all platforms. The default is FALSE. Set to TRUE to enable GlobalCall.
audio.dialogic.Protocol	Set the specific network protocol to use when GlobalCall is in use. The default is "isdn", other protocols include "us_mf_i" for standard North American T1 robbed-bit signalling. New protocols may be added as dialogic provides support for them. Check your release notes for additional supported protocols.

TABLE 22 (CONTINUED): ADDITIONAL NUANCE CONFIGURATION PARAMETERS

Parameter	Operation
Dialogic DSP Parameters (non-Antares)	
audio.dialogic.FullDuplex	Determines whether concurrent playback and recording is allowed (default=TRUE, requiring two DSPs per channel)
Antares-specific Parameters	
audio.dialogic.Antares	See Basic Parameter Settings, page 267
audio.dialogic.EnableTelephony	See Basic Parameter Settings, page 267
audio.dialogic.EnabledDTMFDetector	See Basic Parameter Settings, page 267
audio.dialogic.EnableEchoCancellation	See Basic Parameter Settings, page 267
audio.dialogic.EnableEp	See Basic Parameter Settings, page 267
audio.dialogic.PlaybackPort	See Basic Parameter Settings, page 268
audio.dialogic.CallerPort	See Basic Parameter Settings, page 268
audio.dialogic.PromptPort	See Basic Parameter Settings, page 268
audio.dialogic.OffHook	See Basic Parameter Settings, page 268

Consistent with the Nuance architecture, echo-canceled, endpointed audio data are recognized by the Nuance Recognition Server, running on the same host or another host.

The code that runs on the Antares card has a separate version number from the rest of the Nuance System. The Antares code contained in version 6 is ANT02.

The Nuance Antares firmware improves channel capacity by off-loading the Nuance Recognition Client. This allows more client connections per host processor. In particular, the following computationally expensive functions are performed by the Antares card:

- Echo cancellation
- DTMF detection
- Utterance endpointing

The Antares firmware also provides record and playback capabilities, allowing it to be used with less expensive Dialogic T1 cards. In particular, you can use it with the DTI/240SC card, which has no Dialogic voice processing resources. It is also compatible with other Dialogic SCBUS telephony cards, including the following:

- D/240SC-T1
- D41/ESC (4 port analog card)
- DTI/300SC
- D/300SC-E1

Support for DID Management

The dialogic audio provider provides DID management capability when used with the new multi-channel RecClient, described in "The Multichannel RecClient" on page 80. You can set up the audio provider's DID manager to route incoming calls based in the DNIS. Applications register one or more numbers they want to service using the `audio.Device` and `audio.dialogic.Lines` parameters.

You can register the incoming numbers (that is, the numbers that were dialed) that you want your application to accept using the following syntax:

```
audio.Device=@number_range[ , number_range]
```

The value of *number_range* can be either a single number or a range of numbers separated by a hyphen. You can also specify multiple ranges separated by commas. The ranges cannot include white space. For example:

```
audio.Device=@4730225
```

Accepts calls dialed to 473-0225

```
audio.Device=@4730225-4730228
```

Accepts calls dialed to 473-0225, 473-0226, 473-0227, and 473-0228

```
audio.Device=@4730225,6143000
```

Accepts calls dialed to 473-0225 and 614-3000

You can also specify a special value, @any, to accept any calls for numbers that no other applications are registered for.

Multiple applications that use the same multi-channel RecClient can register for the same number. The call is directed to the first idle application supporting that number that the DID manager finds. If it cannot find an idle application registered for that number, it sends the call to an application registered for any. The call is dropped if it cannot find an application set up to take it.

In certain T-1 configurations, or when lines are in a hunt-group, an application may want to accept a call on any of a number of lines for one or more numbers. Use the `audio.dialogic.Lines` parameter with DID to set up the range of lines which should be monitored for incoming calls:

```
audio.dialogic.Lines=line_range
```

The *line_range* value can be a single line or a range of lines separated by a hyphen. For example:

```
audio.dialogic.Lines=1
```

Uses only line 1

```
audio.dialogic.Lines=1-24
```

Uses lines 1 through 24

```
audio.dialogic.Lines=1,4,5
```

Uses lines 1, 4, and 5

- You must set this parameter whenever you specify an @-prefixed value for the `audio.Device` parameter.

Multiple applications can specify the same value for the `Lines` parameter. The DID manager monitors all lines specified by all running applications.

Basic Parameter Settings

The Antares card is supported as an extension to the Nuance *dialogic* audio provider. You can enable Antares functionality by setting the Nuance parameter

```
audio.dialogic.Antares=TRUE
```

In this mode, playback, recording, echo cancellation, DTMF detection, and endpointing functionality are all handled by the Antares card. Standard Nuance parameters that control or tune these capabilities are passed down to the Antares card.

Specific modes are enabled or disabled as follows:

- Echo cancellation

```
audio.dialogic.EnableEchoCancellation(string)
```

Turns echo cancellation on (TRUE) or off (default=TRUE). Normally, you would leave this turned on.

- DTMF detection

```
audio.dialogic.EnabledTMFDetector(string)
```

Turns DTMF detection on (TRUE) or off (default=TRUE)

On systems where DTMF detection is not the responsibility of the Nuance Speech Recognition System, you may want to turn this off to save processing resources on the Antares card. These systems are also likely to disable telephony support.

- Endpointing

```
audio.dialogic.EnableEp(string)
```

Turns endpointer on or off (default=TRUE). Normally, you would leave this turned on.

- Telephony

```
audio.dialogic.EnableTelephony(string)
```

Turns telephony support on or off. (default=TRUE)

The ability to disable telephony support in the *dialogic* audio provider is new in this release and works only in conjunction with the Antares card support. In

this mode, some other (third party) component in the system is responsible for controlling the call processing. The Nuance System is not normally used to play prompts in this scenario.

When you disable Nuance telephony support, SCBUS time slot information and call disposition information must be passed into the audio provider via the following Nuance parameters:

`audio.dialogic.OffHook(string)`

You should set this parameter to TRUE at call begin and FALSE at call end.

`audio.dialogic.CallerPort(string)`

SCBUS time slot to listen to for the caller's inputs. This can be changed only when `audio.dialogic.OffHook` is FALSE.

`audio.dialogic.PromptPort(string)`

SCBUS time slot to listen to for the outgoing prompts. This is a required input to the echo-cancellation routine. This can be changed when `audio.dialogic.OffHook` is TRUE or FALSE in SCBUS mode. A typical sequence at start of call is:

- i. set `audio.dialogic.PromptPort` to outgoing time slot
- ii. set `audio.dialogic.CallerPort` to incoming time slot
- iii. set `audio.dialogic.OffHook` to TRUE

To change prompt source in mid-call (say from prompt-playback to text-to-speech) set `audio.dialogic.PromptPort` to new source time-slot.

At end of call, set `audio.dialogic.OffHook` to FALSE.

In addition, if the Antares card will be playing the prompt, getting parameter:

`audio.dialogic.PlaybackPort(string)`

will return the SCBUS time slot on which the prompt will be played. It is up to you to route this time slot to the appropriate network connection.

- **GlobalCall**

Preliminary support for Dialogic's new GlobalCall API is available. This API is intended to hide network-specific and country-specific differences

from the application developer. The release notes indicate which platforms and protocols are currently supported.

To activate GlobalCall, set the parameter `audio.dialogic.GlobalCall` to TRUE and specify the protocol to use in the parameter `audio.dialogic.Protocol` (for example, `isdn` or `us_mf_i`).

All Antares Parameters

Antares parameters are described in the following tables:

TABLE 23 : ENDPOINTER PARAMETERS

Parameter	Default	Settable
<code>ep.StartSeconds</code>	0.30 (0.15)	on hook only ^a
<code>ep.EndSeconds</code>	0.75	on hook only
<code>ep.AdditionalStartSilence</code>	0.30	on hook only
<code>ep.AdditionalEndSilence</code>	0.30	on hook only
<code>ep.ThresholdSnr</code>	10	on hook only

a. Default is 0.15 in Nuance parameters database.

TABLE 24: PLAY/RECORD PARAMETERS

Parameter	Default	Settable
<code>client.KillPlaybackOnBargein</code>	TRUE	always

Installation

Follow these procedures to initialize the Dialogic Antares card. You need three files:

- `$NUANCE/antares/antcli.cof` downloaded code file (binary)
- `$NUANCE/antares/antcli.prm` boot-time config file (text)
- `$NUANCE/antares/antares.cfg` example config file

Make the following assumptions

- The Antares drivers and hardware have already been installed. The Nuance distribution does not contain any Dialogic modules or drivers. The Dialogic system release drivers and Antares driver must be obtained

separately from Dialogic. These should be requested from Dialogic when you order the Dialogic hardware.

- The Nuance Recognition System has been installed.

Edit the *antares.cfg* file and the *antcli.prm* file to match your system. The example file */usr/dialogic/antares.cfg* is useful in helping with this process. If the Nuance audio provider will be the only Antares application, then edit */usr/dialogic/antares.cfg* (as root) so that the board parameters and driver parameters match the example file where appropriate.

- ❖ **Caution:** Do not change the interrupt level or port address; these must match your installation.

If there are more Antares boards and other applications, be careful to edit the section(s) for the correct board number(s).

Consider these parameters for each Antares board:

```
Antares_Board = 0      #for board number 0. add another
                        block for additional boards

Port = 0x0200          # probably don't change
MessageSize = 0x80
maxMessageSize = 0x100
maxMessages = 32
maxRouteList = 32
maxMapList = 32
maxBDstreams = 64 # probably need to set this,
                  # because the default is lower
PCMConfig = SCSA      # we require SCBUS mode
MaxSCSAslots = 1024
LineConfig = E1
ClockMode = SYNC
Encoding = Mu_Law

DSP = 0 1 2 3         # all dsps get the same code and
                        parameters

COFF = $NUANCE/antares/antcli.cof
                        #where to find the download code

PARAMFILE = $NUANCE/antares/antcli.prm
                        #where to find the parameters
```

- Note: Convert these to absolute paths for your system. The Dialogic loader will not process environment variables

The following host driver parameters may need to be changed for the Antares host driver configuration:

```
Maximum RCUs allowed in system (default = 12).
Max_Rcus = 32

Maximum opened RCUs (pseudo RCUs) allowed in system
  (default = 12).
Max_Opened_Rcus = 32

Maximum bulk data streams allowed in system (default = 8).
Max_Bulk_Data = 64

Default size of bulk data buffer (default = 0x2000).
This is PC standard memory that is dedicated to Antares
System.
Bulk_Data_Buffer = 6144

Maximum DPI requests allowed in system (default = 4).
Max_Dpi = 64

Maximum length of messages in octets (bytes) (default = 200).
Message_Length = 500
```

You should determine that these parameters do not conflict with any other vendor's configuration if you are using other Antares cards. Choose the maximum value in these cases.

For \$NUANCE/antares/antcli.prm, see the parameter descriptions in the file. As shipped, it supports 16 channels per card and expects to read the prompt input to the echo cancellation routine from an SCBUS time slot.

- Note: The Nuance-Antares client does not currently employ the *dongle* protection scheme.

Native Audio Provider

The native audio provider requires no hardware or software other than what is provided with the platform (except a microphone), but it has no built-in telephony capabilities. On PCs running Solaris/x86, it requires a Sound Blaster card. It is probably the fastest way to prototype and test a speech

recognition application and the associated grammars. Because it is not telephony-based, however, it does not support barge-in (see "Barge-In" on page 331).

The audio parameters listed in Table 25 are specific to the native audio provider. The leftmost column in the list indicates the name of the parameter, whereas the columns on the right indicate the valid and default values for the different platforms. Each item in bold is the default. For more information on the standard audio parameters, see Chapter 14.

TABLE 25: AUDIO PARAMETERS

Parameter Name	sparc-solaris	i386-solaris	win32
audio.Device	/dev/audio /dev/sound/[0-n]	/dev/audio /dev/sound/[0-n]	Runtime-gettable only. This parameter is filled with the name of the first device capable of supporting the requested audio.Format. If the name of the input and output devices are different, the parameter is given the name of the input device.
audio.Format ^a	8k-16bit-linear 8k-8bit-mulaw 16k-16bit-linear 8k-8bit-alaw	8k-16bit-linear 8k-8bit-mulaw 16k-16bit-linear	8k-16bit-linear 16k-16bit-linear 8k-8bit-mulaw 16k-8bit-mulaw
audio.InputVolume	0-255 127	0-255 225	Not supported.
audio.OutputVolume	0-255 175	0-255 175	0-255 67
audio.InputSource	mic line	mic line	mic
audio.OutputDest	speaker headphone line	speaker headphone line	speaker

TABLE 25: AUDIO PARAMETERS

Parameter Name	sparc-solaris	i386-solaris	win32
audio.native. InputPortBehavior	open-on-demand open-right-away open-after-first-use	open-on-demand open-right-away open-after-first-use	open-on-demand open-after-first-use open-right-away
audio.native. OutputPortBehavior	open-on-demand open-after-first-use open-right-away	open-on-demand open-after-first-use open-right-away	open-on-demand open-after-first-use open-right-away
audio.native. InputChunkSeconds	0.05-1.0 0.1	0.05-1.0 0.1	0.05-1.0 0.1

- a. If no format is specified on sparc-solaris or i386 platforms, all audio formats are tried in the order shown until the device is opened successfully. This is done so that older audio hardware on machines such as a SPARC 2 (which support only mulaw encoding) will work without requiring a specified format. The audio format is made up of three components: sampling rate, sample size, and encoding. Any subset of these three components in any order can be specified. The components must be delimited by hyphens, the sampling rate must be followed by the letter k or K, the sample size must be followed by the word bit, Bit, or BIT, and the encoding must be linear, mulaw, or alaw (all lowercase). For example, linear is a valid format and would match the first of the valid formats on the given platform. Other syntactically valid examples are 16bit-linear, mulaw-8bit, and 16bit-8k.

Computerfone Audio Provider

The Computerfone is a single-line analog telephone interface that routes a telephone audio signal to your computer's A/D and D/A converters. It performs call-control communication via an RS-232 serial port interface. The device can be purchased through Nuance Communications and does not require any software other than the Nuance toolkit. The *cfone* audio provider allows the RecClient to control this device.

The Computerfone has five jacks on the back. The IN and OUT jacks hook up to mini-phono cables that transmit and receive analog audio data to and from the workstation. Connect the IN jack on the back of the Computerfone to HEADPHONE-OUT on the workstation, and the OUT jack to LINE-IN on the workstation. Connect the 9-pin serial connector on the back of the Computerfone to a serial port on the back of the workstation. Connect the phone jack to the phone line being used, and the power jack to the Computerfone's power supply.

Because the *cfone* audio provider uses the workstation's built-in audio device, it makes use of the *native* audio provider for playback and recording, adding telephony functionality on top of it.

Pertinent information about configuration parameters is in Tables 26 and 27.

TABLE 26: NOTES ON STANDARD NUANCE CONFIGURATION PARAMETERS FOR THE *cfone* AUDIO PROVIDER

Parameter	Operation
audio.Provider	Must be set to <i>cfone</i> to select this provider
audio.Device	Specifies the underlying native audio provider's device
audio.InputSource	Automatically set to <i>line</i>
audio.OutputDest	Automatically set to <i>headphone</i>
audio.InputVolume	Automatically set to an appropriate platform-dependent value
audio.OutputVolume	Automatically set to an appropriate platform-dependent value

TABLE 27: ADDITIONAL NUANCE CONFIGURATION PARAMETERS FOR THE *cfone* AUDIO PROVIDER

Parameter	Operation
<i>cfone</i> .SerialPort	The UNIX serial device to which the Computerfone is connected (the default for this parameter depends on the platform—on <i>sparc-solaris</i> it is <i>/dev/ttya</i>)

Known Limitations

The `RCPlaceCall()` function is not supported.

Periphonics Platform

The Periphonics VPS platform is, of course, a telephony platform. The Nuance System interfaces with it in a non-telephony-based manner, because the call control, dialog management, and routing of digital audio is done on the Periphonics box itself in the PeriProducer language. The Nuance engine is

considered a peripheral device that gets its audio through what it perceives as a native audio provider on a Sun workstation.

Periphonics may also have implemented additional audio providers, which plug into the Nuance RecClient via shared objects. Contact Periphonics for more information.

IBM DirectTalk Platform

Nuance has implemented a custom recognition client that enables a DirectTalk application to perform speech recognition. The Nuance DirectTalk Custom Client has been implemented below the RCAP, that is, it does not use the audio provider mechanism to acquire digital audio samples.

This implementation does not require all the features the RecClient provides, because the IBM DirectTalk engine handles prompting, DTMF detection, and call control. The Nuance custom client's main purpose is to provide speech recognition functionality to the application.

Voicetek VTK Platform

The Voicetek VTK Platform is a turnkey telephony application platform that can be purchased from Voicetek Corporation. Nuance has integrated its automatic speech recognition (ASR) system with this platform. This means that you can benefit from Nuance's industry-leading ASR technology while developing speech-enabled applications for the Voicetek platform. The Voicetek platform performs call control, DTMF detection, and prompting, while the Nuance software performs speech recognition. Contact a Nuance sales representative for more information.

Writing a Custom Audio Provider

It is possible to add your own audio provider that the Nuance engine can use like any other provider. This requires implementing a set of functions that are audio- or telephony-related (or both) and linking the compiled object file for that implementation into your application. Optionally, you can create a shared object (.so file) that the Nuance engine can load at runtime to access the new audio provider. Shared objects are not available on all UNIX platforms,

however. Contact Nuance technical support for more information about writing your own audio provider.

CALL CONTROL FUNCTIONS

The RecClient programming interfaces (both the RCAPI and the RCEngine) have built-in call control functions that allow you to :

- Hang up—disconnect the current call in progress
- Dial—place an outgoing call
- Answer—establish a connection to an incoming call
- Transfer—transfer the current call in progress to another number

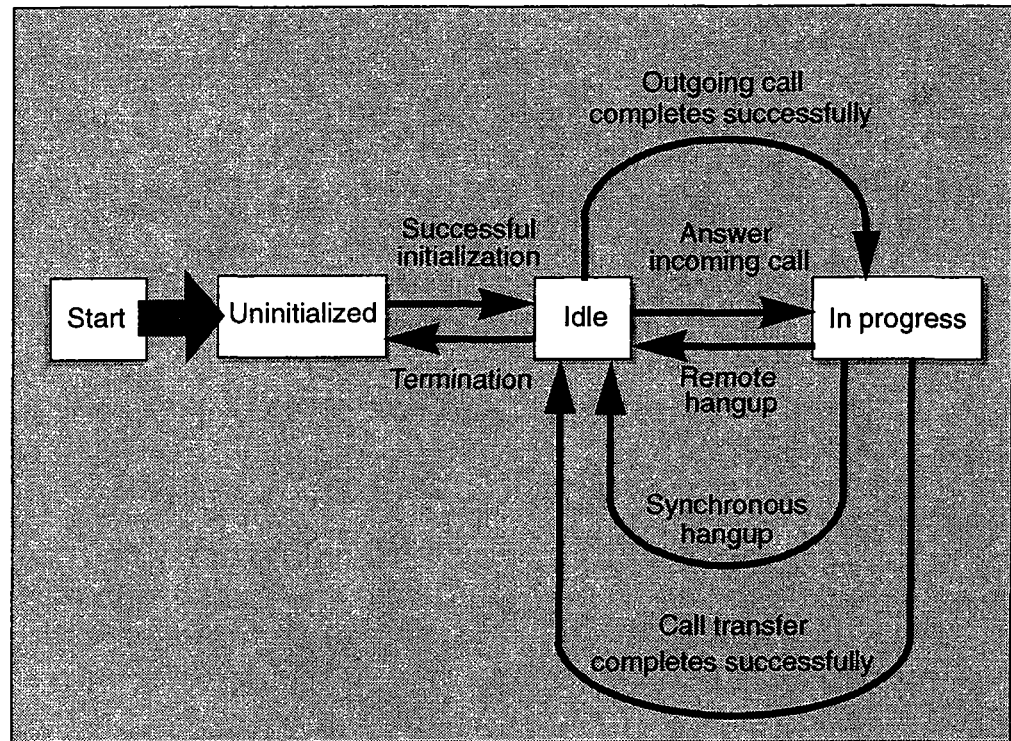
For detailed information on specific functions to call, see Chapter 12, “Programming with the RCEngine,” or Chapter 9, “Programming with the RecClient API.”

TELEPHONY STATES

Conceptually, the RecClient is in one of three states for telephony and call control:

- Uninitialized—before successful initialization and after termination
- Idle—waiting for an incoming call or able to place an outgoing one
- In progress—audio channel exists between the Nuance engine and the user, and recording and playback can occur

Other intermediate states are omitted in this discussion to avoid confusion. Error conditions are not included in the telephony states diagram on page 277.



TELEPHONY EVENTS

The RecClient programming interfaces define several call-control-related Nuance events that can occur asynchronously (for more information, see "RecClient Notification Objects" on page 157 if you are using the RCEngine interfaces, or Chapter 10 if you are using the RCIPI). The telephony events that the Nuance RecClient might send to the application are:

- Remote hangup—the phone at the other end was hung up (will never be generated in the *uninitialized* or idle state)
- Incoming call—a connection is being requested from a remote caller (will be generated only in the idle state)
- Outgoing call completed—the call attempted by either the *dial* or *transfer* command completed successfully (the phone at the remote end was picked up) or unsuccessfully (the remote was busy) (will be generated only following a call to either the *dial* or *transfer* functions)

-
- DTMF—a touch tone was detected (will be generated only in the *in-progress* state)

STAYING FLEXIBLE

The RecClient hides most of the differences between audio providers, such that applications written for one audio provider should work with all audio providers. The major difference between providers that you must be aware of is that some providers support telephony, while others do not.

Applications that use the RCEngine or RCAPI can be written to work with either of the two types of providers somewhat transparently. *Xapp* and *sample-application* are good examples of applications that support both telephony-based and non-telephony-based audio providers.

An application can determine whether the selected audio provider supports telephony by querying the value of the `audio.SupportsTelephony` Nuance parameter.

A telephony-savvy application will query `audio.SupportsTelephony`, and then perform call control functions only if the active provider is in fact a telephony provider.

16

DYNAMIC GRAMMARS

The Nuance System allows the dynamic creation and modification of recognition grammars from within a running application. This capability enables many new types of speech recognition applications, such as:

- A speed dialer, in which speaker-dependent words are associated with phone numbers—a personal speed dial list is maintained for each enrolled user of the system, and the user's own grammar of names (such as "mom", "the office", and "robert jones") is dynamically loaded when that user calls
- A database front end, in which the application looks at the set of currently selected items, and dynamically builds a recognition grammar that allows the user to select from among those items
- Any application in which the items to be recognized are different every time, and cannot be determined until runtime

You can build any of these applications with features of the Nuance System.

DYNAMIC GRAMMAR GSL SYNTAX

You add dynamic grammar capability to an application by specifying a dynamic grammar as a component of a top-level grammar. You define the dynamic grammar in your *grammar specification file*, as follows:

```
GrammarName:dynamic [ ]
```


GrammarName is the *label* that identifies this dynamic slot when you want to modify it during recognition. The keyword `dynamic` indicates that this grammar may be modified at runtime as described here. The brackets indicate an OR construction that is empty until words are added to the grammar at runtime.

- Note: A top-level grammar cannot be dynamic.

The following example of a simple GSL file specifies an order from a breakfast menu that has daily specials:

```
.Entree [ Pancakes Eggs Specials ]
Pancakes [ (blueberry pancakes) (buckwheat pancakes) ]
Eggs [ (fried eggs) (scrambled eggs) ]
Specials:dynamic [ ]
```

When the application is initialized, *Specials* is an empty grammar, so *.Entree* contains only “blueberry pancakes”, “buckwheat pancakes”, “fried eggs”, and “scrambled eggs”. To add the specials to the grammar at runtime, the application assembles a *WordGrammarInfoList* object containing the new words to be recognized, and places this object into the *Specials* grammar.

Dynamic Grammar Runtime API

The Nuance System supplies a data object called a *WordGrammarInfoList* (WGIL—pronounced “wiggle”). A WGIL is a list of words to be assembled into an OR grammar and recognized. API functions exist to create, add to, delete from, read from and write to file, and to delete WGIL objects. These functions are explained and prototyped in the files *\$NUANCE/include/word-grammar-info-list.h* and *\$NUANCE/include/word-grammar-info.h*. The following fundamental actions are involved in recognizing a dynamic list of words:

1. Assemble a WGIL containing the words to be recognized.
2. Insert the WGIL into your grammar at a dynamic grammar location.

The WGIL is a list of word objects, known as *WordGrammarInfo* objects, or WGIs (pronounced “wiggies”). Each WGI contains a word’s name, pronunciation, probability, and natural language statement. To continue the earlier example, suppose you wanted to add “waffles” to today’s breakfast menu.

- The word's name is simply the text string that the recognizer returns when this word is recognized—that is, "waffles".
- The word's pronunciation can be determined by asking the user to speak the word (as described on pages 282 through 285, in the discussion about enrollment) or by looking the word up in the Nuance pronunciation dictionary. A pronunciation might be "w aa f ax l z".
- In most cases, use 1.0 for the word's probability, even if there are multiple words in the WGIL. This setting guarantees that all words will be weighted equally.
- The NL statement associated with the word will be executed if and only if the associated word is recognized, just as if the word and NL statement were written in the grammar file. Legal NL statements might be the empty string "" or "{return(waffles)}".

Once you have constructed a WGIL (which can contain one or more WGIs), you can insert it into your grammar as follows:

```
RCSetWordGrammar(rec_obj, the_wgil, "Specials");
```

Each time you call `RCSetWordGrammar()`, any words previously added to the *Specials* grammar are cleared, and the new WGIL replaces them. Thus, if your grammar file looks like:

```
Specials:dynamic []
. Entree [ pancakes      {return(pancakes)}
           eggs          {return(eggs)}
           Specials:item {return($item0)}
        ]
```

and your WGIL contains the word "waffles" as described above, after calling `RCSetWordGrammar()`, the grammar *.Entree* will behave as if the following had been compiled:

```
. Entree [ pancakes      {return(pancakes)}
           eggs          {return(eggs)}
           waffles       {return(waffles)}
        ]
```

To further modify *Specials*, use the WGIL API to clear, add to, or delete from the WGIL, and then call `RCSetWordGrammar()` again. All management of

dynamic grammars revolves around WGIL objects, which can also be read from or saved to file. More information on WGILs and WGs can be found later in this chapter.

ENROLLMENT: ADDING WORDS BY SPEAKING THEM

One application that requires dynamic grammars is a telephone speed dialer, where callers add and delete names from their personal speed dial lists. In this case, however, names are added by voice, not by text. The Nuance System supports such applications through *enrollment*. With enrollment, the system can listen to the caller speak a word, generate a pronunciation for that word, and return that pronunciation to the application. The application can then include this pronunciation in a WGIL, as described earlier. In the case of a voice dialer, the phone number associated with the newly enrolled word can be stored as the "name" of the new word.

The Nuance System determines the pronunciation for a new word by performing *phonetic recognition* on each example of the word. When performing phonetic recognition, the recognition result is simply an ASCII string representing the phonetic sequence observed.

To perform pronunciation enrollment, include the standard Nuance grammar *enrollment.grammar* in your application grammar file. The *enrollment.grammar* includes a subgrammar called *EnrollmentGrammar*, which you must include in a top-level grammar as follows:

```
; enrollment.grammar defines "EnrollmentGrammar"
#include "enrollment.grammar"

.Enrollment EnrollmentGrammar
```

You can also include the other grammars needed for your application. Compile your grammar file using any model set that supports enrollment. The resulting recognition package can be used for recognition tasks in the standard manner or for generating the pronunciation mappings for an utterance. To learn the pronunciation for a new word, prompt the caller to say the word, and then call `RCRecognize()`, specifying the *.Enrollment* grammar. The recognition engine will generate a pronunciation for the new word, and

return this pronunciation as the recognition result in the `RecResult` structure.

- Note: If N-Best recognition is enabled, multiple pronunciations will be returned in `RecResult`.

An enrolled pronunciation is a string of *phonetic units* (phonetic units are individual speech sounds, such as "sh"). In the Nuance System, these units have names such as "gpm342", where "gpm" stands for Generalized Phone Model. For example, when you select the *.Enrollment* grammar and say "Nuance Communications" the recognizer might return the following phonetic sequence:

```
"gpm812 gpm1044 gpm851 gpm1034 gpm840 gpm1009 gpm1048
gpm727 gpm687 gpm1042 gpm485 gpm959 gpm961 gpm198 gpm811"
```

This string phonetically describes the words you just said. This phone string can then be added to a WGIL, and used as part of a dynamic grammar.

The pronunciation generated is heavily dependent on the characteristics of the given utterance, and in fact is likely to be different each time the same person says it. Thus, it may be considered to be dependent on the speaker of that utterance, and will not necessarily be a good pronunciation for the same phrase from a different speaker. It is recommended that these automatically generated pronunciations be used to recognize utterances only from the speaker who enrolled them.

For recognition robustness, Nuance Communications recommends that applications ask the caller to speak the new word twice, and include both pronunciations in the WGIL.

Enrollment generally takes longer than standard recognition. Enrolled phrases should be kept short (one or two words), so that enrollment can be completed quickly.

- Note: Enrolled pronunciations are model-set specific. Phrases enrolled using one model set cannot be used for recognition with other model sets.

Example of Using Enrollment Grammars

The use of enrollment grammars is demonstrated in the following simple example in which the user first enrolls pronunciations for hello and goodbye and can then test the pronunciations. The grammar is simply the enrollment grammar and a dynamic section:

```
#include "enrollment.grammar"
.Enrollment EnrollmentGrammar
PersonalPronunciations:dynamic []
.Application PersonalPronunciations
```

The skeleton application would be:

```
/*-----*/
/* Get a pronunciation for hello */
/*-----*/
<Prompt for 'hello'>
<Perform recognition with grammar ".Enrollment">
<Get RecResult recresult>
status = RecResultString(result, 0, hello_pron,
    sizeof(hello_pron));

if (status != NUANCE_OK) { <Handle Error> }
/*-----*/
/* "hello" pronunciation should now be in hello_pron */
/*-----*/
/*-----*/
/* Get a pronunciation for goodbye */
/*-----*/
<Prompt for 'goodbye'>
<Perform recognition with grammar ".Enrollment">
<Get RecResult recresult>
status = RecResultString(recresult, 0, goodbye_pron,
    sizeof(goodbye_pron));
if (status != NUANCE_OK) { <Handle Error> }
/*-----*/
/* "goodbye" pronunciation should now be in goodbye_pron */
/*-----*/
/*-----*/
/* Build a WGIL for the pronunciations */
/*-----*/
pronunciation_wgil = WGILNew();
wgi = WGINew("hello");
```

```

WGISetPronunciation(wgi, hello_pron);
WGILAddItem(pronunciation_wgil, wgi);
WGIDelete(wgi);
wgi = WGINew("goodbye");
WGISetPronunciation(wgi, goodbye_pron);
WGILAddItem(pronunciation_wgil, wgi);
WGIDelete(wgi);
/*-----*/
/* Add the pronunciation mappings to the dynamic grammar */
/*-----*/
RCSetWordGrammar(recclient, pronunciation_wgil,
    "PersonalPronunciations");
/*-----*/
/* Test out the pronunciations */
/*-----*/
do {
    <Prompt for test utterance>
    <Do recognition with grammar ".Application">
} while (!end_of_the_world)

```

In this example, only a single pronunciation was generated for each word. In practice, two pronunciations should be generated for each word by asking the user to repeat the name. It is also possible to add natural language interpretations and pronunciation probabilities by using the WGIL structure, as described later.

CLASH TESTING

A problem with allowing pronunciations to be generated by voice is that a user may enroll similar phrases that are intended to have different meanings. This can increase the recognition errors and make an application frustrating to use. A useful test for preventing this problem is the clash test mechanism that is built into the enrollment process. This test is performed as enrollment is taking place, and it returns a status value as to whether the new phrase sounds similar to a previously enrolled phrase.

To activate the clash test, a set of pronunciations against which the test is performed must be passed to the recognition engine. This procedure is achieved by using an approach similar to setting the dynamic grammars. The pronunciations are assembled into a WGIL format and sent to the recognizer via the RecClient using this function:

```
RCSetClashPronunciations(rec_obj, the_clash_wgil)
```

The NL and probability portions of the WGIL are ignored, and so do not need to be explicitly set. The function call must be made before enrollment begins.

After the enrollment result has been returned, the clash status can be obtained from the `RecResult` with:

```
nuance_status = RecResultClashStatus(rec_result,
                                     &clash_status)
```

where `clash_status` is an integer return value. If the clash status is negative, the new phrase does not clash. A non-negative result indicates that the phrase clashed with one of the supplied pronunciations. The clash status gives the index of the pronunciation in the WGIL supplied by `RCSetClashPronunciations` that most closely matched the new phrase.

The sensitivity of the clash test is controlled by the Nuance parameter:

```
rec.ClashThreshold
```

A high threshold means that less similar phrases might clash, and a low threshold means phrases are less likely to clash. The default value for this is 1000. A value of 0 results in no clashes being detected—effectively performing no clash testing. A higher value such as 50000 results in many clashes being detected.

- Note: The `RecServer` automatically clears the clash pronunciations after each enrollment.

CONSISTENCY TESTING

A problem with getting multiple examples of the same phrase for enrollment is that the phrase is not repeated the same way each time. For example, there may be a sudden background noise such as a door slamming, or the user might sneeze during the enrollment. These effects will result in inconsistent pronunciations being generated for the phrases.

To try to catch such problems, a second test on the enrollment utterances can be performed—the consistency test. This test tries to determine that utterances supposed to represent the same phrase do indeed sound alike.

The consistency test is analogous to the clash test. Pronunciations with which the new utterance is supposed to be consistent are passed into the recognizer by using:

```
RCSetsConsistencyPronunciations(rec_obj,
    the_consistency_wgil)
```

Again, the NL and probability portions of the WGIL are ignored, and so do not need to be explicitly set. The function call must be made before enrollment begins.

After the enrollment result has been returned, the consistency status can be obtained from the `RecResult` with:

```
nuance_status = RecResultConsistencyStatus(rec_result,
    &consistency_status)
```

where `consistency_status` is an integer return value. Each bit in the `consistency_status` value encodes the consistency result of a particular pronunciation. If the n th bit is 1, the n th pronunciation in the supplied WGIL is inconsistent with the new example of the phrase.

The WGIL indices are zero based. Thus, a return value of 0 indicates that all pronunciations are consistent, and a return value of 5 (00000101) indicates that the 0th and the second pronunciations are inconsistent, and all others are consistent.

- Note: If more than 32 pronunciations are supplied for consistency, then only the first 32 are used for the consistency test.

The sensitivity of the consistency test is controlled by the Nuance parameter

```
rec.ConsistencyThreshold
```

A low threshold means that phrases must be very similar to be consistent, and a high value means that phrases that are less similar can be consistent. The default value for this parameter is 3500. A value of 0 will make all supplied pronunciations inconsistent, and a value of 50000 will make nearly all pronunciations consistent.

- Note: The `RecServer` automatically clears the consistency pronunciations after each enrollment.

THE WORDGRAMMARINFO API

The *WordGrammarInfo* structure is a single entry in a *WordGrammarInfoList* structure.

A WGI—*word grammar info*—is a container for four attributes: *name*, *pronunciation*, *probability*, and *NL command*. The WGI API has functions that perform the following actions:

- Create/copy/destroy an entire WGI object.
- Set/get a WGI's four attributes—*name*, *pronunciation*, *probability*, and *NL command*.
- Compare one WGI to another.

Creating, Copying, or Destroying a WGI

To create a new WGI, call `WGINew()` with the desired name. Once a WGI has been created, its name cannot be explicitly changed. To use the earlier example, create the “waffles” WGI like this:

```
wgi = WGINew("waffles");
```

If `WGINew()` was successful, `wgi` now contains the name “waffles”, an empty pronunciation string, a probability of 1.0, and an empty NL statement. You can explicitly set and get the latter three attributes, but you can only explicitly get the name, not set it.

A WGI can be copied in either of two ways. The function `WGIClone()` creates a new WGI that is a copy of the original, whereas `WGICopy()` copies a source WGI to a target WGI:

```
wgi2 = WGIClone(wgi);
```

To free up any resources your WGI used, your program should do the following when it is done with the object:

```
WGIDelete(wgi);
```

Setting and Getting WGI Attributes

Setting and getting a WGI's attributes is very straightforward, as shown in the following example:

```
WGISetPronunciation(wgi, "w aa f ax l z");
WGISetNLCommand(wgi, "{return(waffles)}");
wgi2 = WGINew(WGIGetName(wgi));
WGISetPronunciation(wgi2, "w aa f l z");
WGISetNLCommand(wgi2, WGIGetNLCommand(wgi));
```

Comparing One WGI to Another

You can compare WGIs in one of two ways. `WGIIsEqualTo()` simply returns a Boolean value indicating whether all attributes of one WGI are equal to their counterparts in the other one. `WGICompare()` returns a relative value analogous to `strcmp()`, based solely on the names of the two WGIs. It is provided as a convenient compare function for `qsort()` and `bsearch()`.

```
if (WGIIsEqualTo(wgi, wgi2)) {
    printf("The two WGIs are identical.\n");
}

cmp_result = WGICompare(&wgi, &wgi2);

if (cmp_result < 0) {
    printf("wgi's name is lexicographically less than
        wgi2's.\n");
} else if (cmp_result > 0) {
    printf("wgi's name is lexicographically greater
        than wgi2's.\n");
} else {
    printf("wgi's name is lexicographically equal to
        wgi2's.\n");
}
```

THE WORDGRAMMARINFOLIST API

A *WGIL*—*word grammar info list*—is simply an ordered collection of WGIs. Since it uses the `WGICompare()` function to sort the WGIs in the list, they are ordered by name. Multiple WGIs with the same name are allowed—that is, you can have more than one WGI that compares equally to another one.

The WGIL API supports the following sets of operations on a WGIL:

- Create/clear/destroy a WGIL.
- Add/find/delete a WGI to/from a WGIL.
- Load/save a WGIL from/to a file.

Creating, Clearing, or Destroying a WGIL

A WGIL can be created in either of two ways. You can create an empty one, or build a new one from a file. Calling `WGILNew()` will create an empty WGIL, whereas `WGILNewFromFile()` will attempt to create a new WGIL and fill it in from the given file. This is equivalent to calling `WGILNew()` followed by `WGILLoadFromFile()`, described later:

```
wgil = WGILNewFromFile("specials.wgil");
```

If your program needs to reuse an existing WGIL without keeping any of the entries, it can, of course, iterate over all the WGIs, deleting them one at a time. As a convenience, and to avoid programming bugs, the WGIL API provides a single function to do this:

```
WGILFlush(wgil);
```

When your program is done using the WGIL, it should free up the resources it used by calling `WGILDelete()`.

Adding a WGI to a WGIL

The purpose of the WGIL is to keep track of a sorted list of WGIs, which is accomplished through the list management features of the WGIL API.

To add a WGI to a WGIL, call `WGILAddItem()`. This will insert the given WGI at the appropriate location in the WGIL, sorting it based on the result of `WGICompare()`.

- **Note:** A WGIL is allowed to contain multiple WGIs with the same name, so adding a second WGI with the same name does not remove the first one. Instead, it adds a new entry to the WGIL.

Finding a WGI in a WGIL

The WGIL object allows random access to the WGIs it contains. To access a particular WGI in the list, use `WGILGetItem()` with the 0-based index of the WGI of interest. The index must be less than the number of WGIs in the list, which can be determined by calling the function `WGILGetNumItems()`:

```
tmp_wgi = WGILGetItem(wgil, 1);

printf("WGI at index 1 is %s\n", WGILGetName(tmp_wgi));
```

- ❖ **Caution:** The index is not guaranteed to remain the same after you add or delete items from the WGIL.

To find a particular WGI in a WGIL, create a WGI with the name you are trying to locate, and call `WGILFindItem()`. The return value is a 0-based index indicating the location of the found WGI in the list. If a WGI was not matched to the one you are searching for, -1 is returned. If more than one WGI match your search WGI, the index to the first one is returned:

```
search_wgi = WGINew("omelette");
wgi_index = WGILFindItem(wgil, search_wgi);

if (wgi_index < 0) {
    printf("No WGI matching \"%s\" found in WGIL.\n",
        WGILGetName(search_wgi));
    found_wgi = NULL;
} else {
    printf("WGI matching \"%s\" found at index %d.\n",
        WGILGetName(search_wgi), wgi_index);
    found_wgi = WGILGetItem(wgil, wgi_index);
}
WGILDelete(search_wgi);
```

Deleting a WGI from a WGIL

WGIs can be deleted from a WGIL in either of two ways. Calling `WGILDeleteItem()` with the 0-based index of the WGI to delete is the most consistent with the rest of the list management functions. However, for convenience, it is often appropriate to delete all WGIs that match a specific name. For this purpose, use the function `WGILDeleteAllMatchingItems()`. As with the find function, you must

create a pattern WGI to match against when performing the deletion. This function returns the number of WGIs deleted from the list:

```
pattern_wgi = WGINew("donuts");
num_deleted = WGILDeleteAllMatchingItems(wgil,
    pattern_wgi);
printf("%d WGIs matching \"%s\" were deleted.\n",
    num_deleted, WGIGetName(pattern_wgi));
WGILDelete(pattern_wgi);
```

Saving a WGIL to File

The external interface to the WGIL API is supplied through functions that operate on text files. This feature provides a means for persistent and portable storage of WGILs.

To save a binary WGIL object to a file, call `WGILSaveToFile()`. This function will attempt to overwrite or create the specified file, writing the information stored in the WGIL to it.

`WGILSaveToFile()` returns `TRUE` on success or `FALSE` on failure.

Loading a WGIL from a File

To load a WGIL from a file into an existing WGIL object, call `WGILLoadFromFile()` with the name of a WGIL file saved by a call to `WGILSaveToFile()`.

`WGILLoadFromFile()` overwrites any existing entries in the WGIL. There is no convenient way to merge in a file without deleting the existing entries.

`WGILLoadFromFile()` returns `TRUE` on success or `FALSE` on failure.

Final Note

The WGI and WGIL APIs may be replaced by a more generalized dynamic grammar runtime API in a future version of the Nuance System.

17

SPOKENSQ

SpokenSQL is the Nuance tool for generating SQL queries. SpokenSQL generates these queries from the interpretations produced by the natural language system. To enable this capability, you must provide certain specifications at compile time in a file called *<package>.sql* in the same directory as the grammar file. These specifications consist for the most part of IF-THEN rules that map filled slots into pieces of SQL. At runtime, these rules are applied, and the various pieces of SQL are assembled into a complete SQL statement.

USE IN APPLICATIONS

SpokenSQL is most useful in applications in which the SQL can be fairly complex. It may not be needed in applications in which the SQL is straightforward. If the queries in your application never have any joins or embedded SELECT statements, you probably do not need to use this tool.

To use SpokenSQL, you must already be familiar with SQL and with the Natural Language System. For information on this system, see Chapters 4 and 5. It is beyond the scope of this manual to provide an introduction to SQL; however, your local bookstore should have books that do.

The SpokenSQL tool merely generates SQL queries; it does not execute them. For information on the Nuance interface to a relational database (through which you can execute queries), see page 120. You may also be able to

interface directly to a database from your application code without using the Nuance interface.

A sample application that makes use of SpokenSQL can be found in `$NUANCE/sample-applications/travel`. This application handles requests for flight information, much like a human travel agent would. It is very useful to look at this application while you are learning about SpokenSQL. The examples in this chapter will also be in the flight information domain.

To test the sample application, or to test your own package, you can use *nl-tool* with the `-sql` option:

```
nl-tool -package travel -sql
```

This option causes *nl-tool* to generate and display SQL queries for the sentences you type. See page 49 for more information on *nl-tool*.

A detailed description of the syntax of the SpokenSQL specifications file can be found in Appendix D. This is useful primarily for reference, and not as a learning aid.

Throughout this chapter, text that the developer would enter into the SpokenSQL specifications file is displayed in *courier* font. SQL queries that the system might output are displayed in *helvetica* font.

API FUNCTIONS

As mentioned, the rules that map filled slots into SQL are specified by the developer at compile time. These rules are placed in a file named `<package>.sql` (where “package” is the name of the package) and will automatically be compiled by *nuance-compile* or *nl-compile*, with the resulting binary `sql_info` file being placed into the package directory.

If you are building your application with the Dialog Builder, the only API function you need is the following:

```
NuanceStatus AppGenerateSQL(App *app, NLResult *nl_result,
    char *query, int buffer_length);
```

This function generates the query from the active interpretation of the given `NLResult` object. The query produced is placed in the *query* buffer, provided that it fits. `NUANCE_OK` is returned if there are no problems. If there is an

ambiguity in the rules, and more than one query could be produced, one query is placed into the buffer, and NUANCE_AMBIGUITY is returned. Other status codes (such as NUANCE_COULD_NOT_PRODUCE_SQL_QUERY) may also be returned. It is important to check the returned status code in your application, because not every error in the *.sql* file can be caught at compile time.

If you are not using the Dialog Builder, there is a separate API. You will need to create and manipulate two objects, the `SQLEngine` and the `SQLResult`. These can be initialized and disposed of via the following functions:

```
SQLEngine *SQLInitializeEngine(NuanceConfig *config,
                               NuanceStatus *status)

SQLEngine *SQLInitializeEngineFromPackageDir
(char *package, NuanceStatus *status)

void SQLFreeEngine(SQLEngine *sql_engine)

SQLResult *SQLInitializeResult(NuanceStatus *status)

void SQLFreeResult(SQLResult *sql_result)
```

The following API function actually generates the SQL query or queries from a given interpretation:

```
NuanceStatus SQLCreateQueries(NLResult *nl_result,
                              SQLEngine *sql_engine, SQLResult *sql_result)
```

`SQLGetIthQuery()` gets the *i*th SQL query (the actual text) out of an `SQLResult` object:

```
NuanceStatus SQLGetIthQuery(SQLResult *sql_result,
                             int i, char *buffer, int buffer_length)
```

Remember that in certain cases multiple queries may be produced, if the rules in the *.sql* file are ambiguous. To get all the queries produced, call `SQLGetIthQuery()` with progressively larger values of *i* until `NUANCE_ARGUMENT_OUT_OF_RANGE` is returned.

In many cases, an application wants to generate SQL from the slots filled over the course of a dialog, and not merely from the slots filled in a single utterance. By means of the function `UpdateCumulativeInterpretation()`, you can maintain a cumulative

NLResult object that saves the slots filled during the dialog, and then pass that object to `AppGenerateSQL()` or `SQLCreateQueries()`. See Chapter 8 for information on this function.

A SIMPLE EXAMPLE

Let's suppose that we are building a flight information application, and the database to which we want to interface has a *flight* table with the following structure:

flight_id	airline	number	dept_time	arr_time	from	to
10001	United	123	1630	2000	BOS	SFO
10002	Delta	456	1030	1930	LAX	JFK
.						
.						
.						

Suppose further that we have written our grammar so that the following utterance produces the interpretation shown here:

"Tell me all the flights leaving after two thirty PM"

{<select_table flight> <dept_after 1430>}

The SQL that we want to send to the database is:

```
SELECT * FROM flight WHERE flight.dept_time > 1430
```

If we provide the following two rules in our SpokenSQL specifications file at compile time, the above query will be generated.

```
IF <select_table flight> THEN SELECT * FROM flight
```

```
IF dept_after THEN WHERE flight.dept_time > $dept_after
```

The first rule says that if the *select_table* slot is filled with the value *flight*, then select all columns from the *flight* table. The second rule says that if the *dept_after* slot is filled (with any value whatsoever), then add the condition *flight.dept_time > \$dept_after* to the WHERE clause with *\$dept_after* replaced by the value in the *dept_after* slot.

As these examples illustrate, the syntax of the rules in the SQL specifications file should be familiar. Many elements are drawn from the syntax of SQL, while other aspects are borrowed from the syntax of the natural language system.

VALUES AND EXPRESSIONS

The values and expressions that can appear within statements in the SpokenSQL specifications file are described here. The various statement types are described on page 302.

An *expression* is something that is either true or false; for example, expressions occur on the left side of IF statements. A *value*, in contrast, usually has an integer or string value. Unlike some programming languages, such as C, values and expressions are not interchangeable. Where you can use one, you cannot use the other.

WHERE Values

In the previous discussion, we saw a WHERE statement with two types of value:

flight.dept_time

\$dept_after

The value *flight.dept_time* is an example of a column reference. All column references in a WHERE statement must have the *table.column* syntax; that is, a column name alone is not allowed in a WHERE statement.

\$dept_after is an example of a slot value reference. At runtime, a slot value reference is replaced by the value of the slot. If you expect the slot to be filled with a structure, you can also refer to the value of a particular feature within the structure. For example, if there were a *dept_time_constraint* slot, which could contain a feature called *after*, then an interpretation might look like this:

```
{<select_table flight> <dept_time_constraint
  [<after 1430> <before 1700>]>}
```

and the following rule would make sense:

```
IF dept_time_constraint THEN WHERE flight.dept_time >
    $dept_time_constraint.after
```

Slot value references can also be combined in arithmetical expressions. For example:

```
IF dept_around THEN WHERE
    flight.dept_time > $dept_around - 15
    AND flight.dept_time < $dept_around + 15
```

This rule specifies that if the *dept_around* slot is filled, then the *dept_time* column of the *flight* table must be within 15 minutes of the time value specified in that slot. As well as addition and subtraction, SpokenSQL supports multiplication (*), division (/), and unary negation (-). You can also use parentheses to indicate grouping.

In the above rule, we saw the use of the integer constant 15 as a WHERE value. String constants can also be WHERE values, providing that they are enclosed in single quotes:

```
IF <airline United> THEN WHERE
    flight.airline_code = 'UA'
```

WHERE Expressions

We have already seen a few examples of WHERE expressions, such as:

```
flight.dept_time > $dept_after

flight.dept_time > $dept_around - 15 AND
    flight.dept_time < $dept_around + 15
```

The first example is a simple comparison expression involving a column and a slot value. Comparison operators, besides ">", are "<", ">=", "<=", and "=". The second example is a Boolean combination of comparison expressions. In addition to AND, the Boolean operators are OR and NOT. You can also use parentheses to indicate grouping.

The SQL comparison operators IS NULL and IS NOT NULL are also supported. For example, suppose the *flight* table has a *meal* column that has the name of the meal served, if any, and is NULL otherwise. Assuming the *meal* slot is filled with *any*, if the user asks for flights serving a meal, the following IF-THEN rule would be appropriate:

```
IF <meal any> THEN WHERE flight.meal IS NOT NULL
```

IF Expressions

The left side of IF-THEN rules can contain all the WHERE expressions and a few other types that cannot appear in WHERE statements. On page 298, we saw examples of these two types of expression:

```
<select_table flight>
```

```
dept_after
```

Slot-value expressions, like the first example, are true when a slot is filled with a particular value. Slot expressions, like the second example, are true when a slot is filled with any value. These expressions can be combined in Boolean combinations just like WHERE expressions.

It is sometimes useful to have slot-value expressions that access a feature within a structure that fills a slot. For example, suppose the origin slot is filled with a structure that contains an airport or city code and a feature indicating the type of code (either airport or city). Possible values for the origin slot would include:

```
[<type airport> <code JFK>]
```

```
[<type city> <code NYC>]
```

The following rule would allow translation of the *origin* slot:

```
IF <origin [<type airport>]> THEN WHERE flight.from
= $origin.code
```

This rule says that if the *origin* slot is filled with a structure that contains *at least* the feature-value pair *<type airport>*, then the constraint shown on the right side should be added to the WHERE clause. Note that the origin slot can (and will) contain other features besides *type*, and yet the expression will still evaluate to TRUE.

STATEMENTS

The SELECT, WHERE, and IF statements may appear alone, as well as embedded in IF-THEN rules. In this case, they are executed unconditionally. For example, if your specifications file has the statement:

```
SELECT * FROM flight
```

then every query produced at runtime will select all columns from the *flight* table.

Statements may be grouped in statement lists. This is useful, for example, in the right side of IF statements:

```
IF <cheap yes> THEN ( SELECT * FROM fare
                      WHERE fare.cost < 200 )
```

This rule says that if the *cheap* slot is filled with *yes*, then execute both the given SELECT statement and the given WHERE statement.

SELECT Statements

As we have seen, a SELECT statement specifies that the basic query type is SELECT, and specifies the SELECT list and the FROM list:

```
SELECT * FROM flight
```

The syntax of this statement is essentially identical to the corresponding SQL syntax. In addition, to "*" you can have specific columns in the SELECT list:

```
SELECT airline, number FROM flight
```

You can also have multiple tables in the FROM list (in which case the column names must have the appropriate table name prefixed):

```
SELECT flight.airline, flight.number, fare.cost
       FROM flight, fare
```

If multiple SELECT statements execute, their results are combined in a natural way into a single SQL SELECT query. For example, if the following statements are both executed:

```
SELECT * FROM flight
```

```
SELECT * from fare
```

then the SQL produced contains both tables in the FROM list:

```
SELECT * FROM flight, fare ...
```

If the following two statements are both executed:

```
SELECT dept_time FROM flight
```

```
SELECT arr_time FROM flight
```

then the SQL produced contains both columns in the SELECT list:

```
SELECT dept_time, arr_time FROM flight ...
```

It is possible for runtime errors to occur if incompatible SELECT statements are executed. For example, you cannot select "*" from one table, and specific columns from another. You also cannot select from two tables if the system does not know how to join the two tables (see page 307 for information on joins).

Where

WHERE statements determine the WHERE clause of an UPDATE or DELETE statement as well as a SELECT query (see page 306 for more information on UPDATE and DELETE statements). We have already seen several examples of WHERE statements, including:

```
WHERE flight.dept_time > $dept_after
```

This statement says to add the condition *flight.dept_time > \$dept_after* to the WHERE clause with *\$dept_after* replaced by the value in the *dept_after* slot.

In general, any number of WHERE statements may be executed for a given interpretation. In the simplest cases, the WHERE clause is constructed by gluing together the various pieces with ANDs. For example, suppose our IF-THEN rules include:

```
IF dept_after THEN WHERE flight.dept_time > $dept_after
```

and:

```
IF airline THEN WHERE flight.airline = $airline
```

Given the following interpretation:

```
{<dept_after 1430> <airline United>}
```

the WHERE clause that will be produced is:

```
WHERE flight.dept_time > 1430 AND flight.airline = 'United'
```

The system may also need to construct a join to link a constraint on one table to the table being selected from. See page 307 for information on joins.

A WHERE statement cannot make reference to a slot that is not guaranteed to be set. For example, the following statement would be flagged as an error at compile time if it did not occur within an IF statement:

```
WHERE flight.dept_time > $dept_after
```

By putting this statement within an IF statement that guarantees that *dept_after* is filled, we avoid this problem:

```
IF dept_after THEN WHERE flight.dept_time > $dept_after
```

This is actually a general constraint on all SpokenSQL statements, not just a constraint on WHERE statements.

Remember that SpokenSQL requires the use of the *table.column* syntax within WHERE statements. For example, this is disallowed:

```
IF airline THEN WHERE airline = $airline
```

while this is fine:

```
IF airline THEN WHERE flight.airline = $airline
```

If-Then

We have already seen the basic structure of the IF-THEN rule:

```
IF some-expression THEN some-statement
```

The syntax of expressions was already discussed on page 301; the various statement types are discussed here.

An IF-THEN rule may contain ELSE clauses. For example:

```

IF <select_table airline> THEN
    SELECT * FROM airline
ELSE IF <select_table fare> THEN
    SELECT * FROM fare
ELSE SELECT * FROM flight

```

This IF-THEN rule says that if the *select_table* slot is filled with *airline* to select all columns from the *airline* table; if it is filled with *fare* to select all columns from the *fare* table. Otherwise, all columns are selected from the *flight* table.

IF-THEN rules may be nested, and statement lists may be used to indicate the attachment of ELSE clauses:

```

IF <type select> THEN (
    IF <table flight> THEN
        SELECT * FROM flight
    ELSE IF <table fare> THEN
        SELECT * FROM fare
    )
ELSE IF <type insert> THEN ...

```

Without the parentheses the ELSE IF <type insert>... clause would attach to the embedded IF statement, and not the top-level one.

Insert Statements

An INSERT statement will be produced if a statement like the following executes:

```

IF <action book> AND name AND airline AND number THEN
    INSERT INTO reservations (name, airline, number)
    VALUES ($name, $airline, $number)

```

Suppose the given interpretation is:

```

{<action book> <name John_Doe> <airline United>
 <number 123>}

```

Then the SQL produced is:

```

INSERT INTO reservations VALUES ('John_Doe', 'United', 123)

```

Just as in SQL, the list of columns is optional; for example, the following is also valid:

```
IF <action book> AND name AND airline AND number THEN
    INSERT INTO reservations VALUES ($name, $airline,
    $number)
```

Remember that slots referred to must be guaranteed to be set. Hence, the following is disallowed:

```
IF <action book> THEN
    INSERT INTO reservations VALUES ($name, $airline,
    $number)
```

Update Statements

An UPDATE statement is produced if a statement like the following executes:

```
UPDATE reservations SET name = 'John Smith'
```

The WHERE portion of the UPDATE statement is produced in the same way as the WHERE portion of a SELECT statement.

Delete Statements

A DELETE statement will be produced if a statement like the following executes:

```
DELETE FROM reservations
```

The WHERE portion of the DELETE statement is produced in the same way as the WHERE portion of a SELECT statement, as described on page 302.

Order By Statements

An ORDER BY clause will be added to the SELECT query being built if a statement like one of the following executes:

```
ORDER BY flight.dept_time
ORDER BY flight.dept_time ASC
ORDER BY flight.dept_time DESC
```

Note that the column must be expressed with the *table.column* syntax.

JOINS

This section describes how to declare the joins in your database, and how this enables the generation of complex embedded SQL queries. As an example, recall our hypothetical *flight* table with the following structure:

flight_id	airline	number	dept_time	arr_time	from	to
10001	United	123	1630	2000	BOS	SFO
10002	Delta	456	1030	1930	LAX	JFK
.						
.						
.						

Suppose we also have a *fare* table that looks like this:

flight_id	class	one_way_cost	round_trip_cost
10001	COACH	150	225
10001	FIRST	325	500
.			
.			
.			

Suppose further that the following utterance:

"Tell me all the fares whose one way cost is less than three hundred dollars."

produces the following interpretation:

```
{<select_table fare> <one_way_limit 300>}
```

The rules we need to handle this interpretation are nothing special:

```
IF <select_table fare> THEN SELECT * FROM fare
```

```
IF one_way_limit THEN WHERE fare.one_way_cost <
    $one_way_limit
```

The SQL that will be produced is:

```
SELECT * FROM fare WHERE fare.one_way_cost < 300
```

However, suppose the utterance and interpretation are:

"Tell me all the flights whose one way cost is less than three hundred dollars."

```
{<select_table flight> <one_way_limit 300>}
```

The rules for the translation of *<select_table flight>* and *<one_way_limit 300>* will fire, but without further specification the system will have no knowledge of how to apply a constraint on the *one_way_cost* column of the *fare* table to a SELECT statement operating on the *flight* table. What the system needs is information on how to join the *flight* and the *fare* tables. This is supplied via the following JOIN statement:

```
JOIN flight.flight_id = fare.flight_id
```

Given this specification, when the system comes to translate the *one_way_limit* slot, it can determine how to join the *flight* and *fare* tables. There are two ways this might be done, depending on whether we are selecting just from the *flight* table or from both the *flight* and *fare* tables. Let's suppose the latter, which would be the case if the rule for the *one_way_limit* slot was the following:

```
IF one_way_limit THEN ( WHERE fare.one_way_cost <
    $one_way_limit SELECT * FROM fare )
```

and the rule for the *select_table* slot was this:

```
IF <select_table flight> THEN SELECT * FROM flight
```

The SQL that would be produced is the following:

```
SELECT * FROM flight, fare WHERE flight.flight_id = fare.flight_id AND
    fare.one_way_cost < 300
```

As you can see, the condition in the JOIN statement is copied directly into the SELECT statement.

Suppose, though, that we are only retrieving data from the *flight* table; that is, the rule for *one_way_limit* is the following:

```
IF one_way_limit THEN WHERE fare.one_way_cost <
    $one_way_limit
```

In this case, the SELECT statement produced would be

```
SELECT * FROM flight WHERE flight.flight_id IN ( SELECT flight_id FROM fare
WHERE fare.one_way_cost < 300 )
```

Instead of adding an equality condition to the WHERE clause the system builds an embedded subquery that retrieves the appropriate flight ids from the *fare* table. In general, the system will automatically construct the necessary subqueries to join a table referenced in WHERE clauses to a table referenced in the FROM list. The example above works because the system knows how to join the *flight* and *fare* tables, and because there was only one way to do that join. You can use named joins to handle situations where this uniqueness condition is not fulfilled.

Named Joins

The *flight* table on page 307 has an origin and a destination column containing airport codes. It is therefore a relatively simple matter to handle utterances asking for flights from or to a given airport. However, suppose we want to handle requests for a flight from or to a given *city*:

"Tell me the flights from San Francisco"

"Tell me the flights to Boston"

Because an airport can serve more than one city, we need a new *airport_service* table in our database to link airports to the cities they serve:

Airport	City
SFO	San Francisco
SFO	Oakland
OAK	San Francisco
OAK	Oakland
JFK	New York
BOS	Boston

Note, for example, that SFO serves both San Francisco and Oakland.

Assume that the sentences have the following interpretations:

```
{<select_table flight> <origin San Francisco>}
```

```
{<select_table flight> <destination Boston>}
```

The rules we need for translating the *origin* and *destination* slots are these:

```
IF origin THEN WHERE airport_service.city = $origin
```

```
IF destination THEN WHERE airport_service.city =  
$destination
```

To apply these rules in building queries on the *flight* table, the system must know how to join the *flight* and *airport_service* tables. The following JOIN statements would seem to accomplish this:

```
JOIN flight.from airport_service.airport
```

```
JOIN flight.to airport_service.airport
```

The problem is that in building a query for the interpretation with the *origin* slot filled, the system does not know which join to use. In other words, without further information the system does not which of the following two queries to produce:

```
SELECT * FROM flight WHERE flight.from IN (SELECT airport FROM  
airport_service WHERE airport_service.city = 'San Francisco')
```

```
SELECT * FROM flight WHERE flight.to IN (SELECT airport FROM airport_service  
WHERE airport_service.city = 'San Francisco')
```

The solution is to name the joins, which can be done by prefacing the JOIN statement with a label and a colon:

```
orig-join: JOIN flight.from = airport_service.airport
```

```
dest-join: JOIN flight.to = airport_service.airport
```

Then, in the rules for translating the origin and destination slots, we indicate which join to use by means of a VIA clause:

```
IF origin THEN WHERE airport_service.city = $origin VIA  
orig-join
```

```
IF destination THEN WHERE airport_service.city =  
$destination VIA dest-join
```

Indirect Joins

The *airport_service* table we have discussed looks like this:

Airport	City
SFO	San Francisco
SFO	Oakland
OAK	San Francisco
OAK	Oakland
JFK	New York
BOS	Boston

A weakness of this table is that it cannot handle pairs of cities that have the same name, but that are distinct; for example, Kansas City, Kansas and Kansas City, Missouri. A better approach would be to have an *airport_service* table that associates airport codes with unique city codes, and a *city* table that associates city codes with city and state names. The *airport_service* table would look like this:

Airport	City
SFO	SANF
SFO	OAKL
MCI	KANK
MCI	KANM
.	
.	
.	

The *city* table would look like this:

city_code	city_name	state
KANK	Kansas City	KA
KANM	Kansas City	MO

city_code	city_name	state
SANF	San Francisco	CA
OAKL	Oakland	CA
.		
.		
.		

The question then is how to handle an utterance like:

"Tell me all the flights from Kansas City, Kansas"

assuming the interpretation is:

```
{<select_table flight> <origin [<city Kansas City>
                                <state KA>]>}
```

The SELECT statement we want to be produced is the following:

```
SELECT * FROM flight WHERE flight.from IN (SELECT airport FROM
airport_service WHERE airport_service.city IN (SELECT city_code FROM city
WHERE city.city_name = 'Kansas City' AND city.state = 'KA' ) )
```

This SELECT statement makes use of two joins: one between the *flight* and *airport_service* tables via the *from* and *airport* columns, and one between the *airport_service* and *city* tables via the *city* and *city_code* columns. Alternatively, we might say that there is an indirect join between the *flight* and the *city* tables, mediated by the *airport_service* table. Any indirect join consists of a sequence of direct joins, which we will refer to as the join path.

For the above SELECT statement to be generated by the system, we need to declare both of the direct joins:

```
orig-join: JOIN flight.from = airport_service.airport
JOIN airport_service.city = city.city_code
```

The WHERE rule we then need is just the following:

```
IF origin THEN WHERE city.city_name = $origin.city AND
city.state = $origin.state VIA orig-join
```

The system can infer the join path needed to generate the SELECT query just on the basis of this WHERE rule and the two JOIN statements. The VIA

statement tells the system to use the *origin* join between the *flight* and *airport_service* tables, and the system is smart enough to figure out that there is one and only one way to join the *airport_service* and the *city* tables.

If there are multiple ways to construct a join path needed in building a given query, the system will construct multiple queries, one for each possible join path. This is one reason the API allows for the possibility of multiple queries being produced, as discussed on page 297. Note that the system will not automatically prefer shorter join paths over longer ones. If you want to eliminate ambiguity, you must do so by means of VIA clauses or with “preferred join paths”.

Preferred Join Paths

If there are two possible join paths between a pair of tables, and one should always be used rather than the other in constructing queries, there is a quicker way to express this than using VIA statements in every rule that requires this indirect join. Suppose the *flight* and *fare* tables both have a *days_code* column that indicate what days of the week each flight or fare is available:

The *flight* table:

flight_id	from	to	days_code	...
10001	SFO	JFK	SA SU	...
10002	SFO	JFK	MO TU WE	...
.				
.				
.				

The *fare* table:

flight_id	class	one_way_cost	round_trip_cost	days_code
10001	Coach	300	450	SA
10001	First	600	850	MO TU
.				
.				
.				

Suppose further that there is a *days* table that maps day codes to individual days of the week:

days_code	day
SA SU	SA
SA SU	SU
MO TU WE	MO
MO TU WE	TU
MO TU WE	WE
.	
.	
.	

We would define the following joins:

```
JOIN flight.days_code = days.days_code
JOIN fare.days_code = days.days_code
```

The problem now is that there are two possible ways of joining the *flight* and *fare* tables: the correct direct join via the *flight_id* column and an incorrect indirect join via the *days* table. In other words, the system has no way of knowing that the following SELECT statement is not the correct query for the following interpretation:

```
{<select_table flight> <one_way_limit 300>}
SELECT * FROM flight WHERE flight.days_code IN ( SELECT days_code
FROM days WHERE days.days_code IN ( SELECT days_code FROM fare
WHERE one_way_cost < 300 ) )
```

We could solve this problem by means of a VIA clause in the translation rule for the *one_way_limit* slot, but this would not be sufficient. We would also need to add such a clause in the rule for the *round_trip_limit* slot, and many other slots. A better approach is to give the join between the *flight* and the *fare* table a name:

```
flt-fare-join: JOIN flight.flight_id = fare.flight_id
```

and then define a preferred join path, as follows:

```
PREFERRED-JOIN-PATH(flt-fare-join)
```

This indicates that the preferred way to join the *flight* and *fare* tables is via the one-step join path consisting solely of *flt-fare-join*. We could also (incorrectly) have made the other join path the preferred one, as follows:

```
flt-days: JOIN flight.days_code = days.days_code
fare-days: JOIN fare.days_code = days.days_code
PREFERRED-JOIN-PATH(flt-days, fare-days)
```

Non-Equi Joins

So far we have only seen examples of equi joins, that is, joins involving the equality of one column and another. However, many other types of joins are possible.

Suppose there is an interval table that associates periods of the day (like morning) with their start and end times:

Period	Begin	End
morning	0	1159
afternoon	1200	1759
evening	1800	1959
night	2000	2359

And suppose the utterance below produces the interpretation shown:

"Tell me the flights departing in the afternoon."

```
{<select_table flight> <dept_period afternoon>}
```

The translation rule for the *dept_period* slot is simply:

```
IF dept_period THEN WHERE interval.period = $dept_period
```

What's left is to specify how the *flight* and *interval* tables are joined. The following JOIN statement does that:

```
JOIN interval.begin <= flight.dept_time AND
      interval.end >= flight.dept_time
```

The query that will be produced is:

```
SELECT * FROM flight WHERE flight.dept_time >= (SELECT begin FROM interval
      WHERE interval.period = 'afternoon') AND flight.dept_time <= (SELECT end
      FROM interval WHERE interval.period = 'afternoon')
```

As you can see, each reference to a column in the *interval* table is converted into an embedded SELECT statement.

The conditions in a JOIN statement can be arbitrarily complicated. The main restriction is that each atomic comparison in a JOIN condition (e.g., `interval.begin <= flight.dept_time`) must refer to each of the two tables being joined. Furthermore, one table must be referenced on one side of the comparison operator and the other table must be referenced on the other side.

Another possibility is to define a JOIN with the “`!=`” operator rather than the “`=`” operator:

```
JOIN flight.flight_id != fare.flight_id
```

If this is done, subqueries will be embedded with “`=`” rather than with `IN`. For example, the SQL produced for:

```
{<select_table flight> <one_way_limit 300>}
```

would be:

```
SELECT * FROM flight WHERE flight.flight_id = (SELECT flight_id FROM fare
      WHERE fare.one_way_cost < 300)
```

instead of:

```
SELECT * FROM flight WHERE flight.flight_id IN (SELECT flight_id FROM fare
      WHERE fare.one_way_cost < 300)
```

18

NUANCE TOOLS: WHAT'S AVAILABLE

The Nuance System includes tools and utility programs that make life easier for the application developer. Some of these tools are briefly described here. In most cases, the purpose is simply to make you aware of the available resources, and to provide a pointer to more information.

SAMPLE APPLICATIONS

The Nuance System is shipped with source code for three sample applications you can use to learn about the system. You can also use these applications as a starting point for your own applications.

Banking

`$NUANCE/sample-applications/banking/src` contains a sample banking application built with the Dialog Builder. This application lets you check your balance, transfer money, pay bills, and add new payees. The “add payee” feature illustrates the enrollment and dynamic grammar capabilities of the Nuance software. Everything you need to build and run the application is provided, including the source code, grammar, dictionary, prompts, Makefile, and scripts.

To run the application, start the server with the `go.recserver` script, and when the server is ready, start the application with the `go.app` script. You will need to

edit the *go.SHARED* script to configure the application for your environment (at the very least, you will need to set *REC_HOST* to the machine on which you are running the *recserver*). You should run the *go.app* script with an argument indicating the audio provider: either *-mic*, *-cfone*, or *-dialogic*.

Xapp

The *Xapp* graphical test application loads any recognition package, determines the names of the recognition grammars in that package, and makes those grammars available via a pull-down menu. Invoke *Xapp* as follows:

```
% Xapp -package recognition_package
```

where *recognition_package* was compiled using *nuance-compile*.

The *Xapp* application also supports dynamic grammars. If one of your recognition grammars contains a dynamic grammar named *DynaGram*, *Xapp* will insert a WGIL into that grammar if you specify the *-add_grammar* argument as follows:

```
% Xapp -package recognition_package -add_grammar
    mygram.wgil DynaGram
```

After initialization, *Xapp* will load the WGIL file *mygram.wgil* and call *RCSetWordGrammar()* to place the WGIL in the dynamic grammar named *DynaGram*. A WGIL file is generated by the command *WGILSaveToFile()*, which is described in Chapter 16.

The source code for *Xapp* is included with your Nuance System distribution, in the directory *\$NUANCE/Xapp/src*.

- Note: Some of the code for *Xapp* is auto-generated.

sample-application

Like *Xapp*, *sample-application* demonstrates how to initialize and run a recognition system. In contrast to *Xapp*, *sample-application* is a text-based program.

The *sample-application* tool also supports the *-add_grammar* feature.

The source code for *sample-application* is included with your Nuance System distribution, in the directory `$NUANCE/src`.

OTHER TOOLS

Xwavedit

The Nuance System includes a waveform editor, *Xwavedit*, which makes it easy to record, play, view, crop, cut, and paste waveforms. The *Xwavedit* program is ideal for recording prompts and removing trailing silence from them. For full details on *Xwavedit*, see the online documentation.

resource-locks

To avoid contention between Nuance System processes for various physical resources, Nuance Communications has implemented a file-based locking scheme. On Unix platforms, you can use the script *resource-locks* to see which resources are currently in use, and by whom.

Currently resource locking is only used by Nuance audio providers, to assure that only one RecClient attaches to each physical telephone line or audio device. If the RCAPI function `RCInitialize()` fails with a return status of `NUANCE_TELEPHONY_DEVICE_BUSY` or `NUANCE_AUDIO_DEVICE_BUSY`, this indicates that some other process is already using the requested audio/telephony resource. To find out which process currently has the resource, use the shell script *resource-locks*, provided with your Nuance System distribution in the `$NUANCE/scripts` directory. For each locked resource, *resource-locks* will print the resource name, followed by the username, ID, and name of the process that is locking the resource at that time.

The output of *resource-locks* might look like this if user "john" was running the "demo-dialer" over a Computerfone:

RESOURCE	USER	PID	COMMAND
audio:_native_	john	8344	demo-dialer
cfone-	john	8338	demo-dialer

The *resource-locks* script also accepts a single option, which is a filter to apply to the resources in use. For instance, to list only resources including the word “cfone”, run

```
% resource-locks cfone
```

Playing Files

The Nuance System includes a simple program *playwav*, which plays audio files through any supported audio provider. This is useful for listening to recorded prompts or to recordings made by a running application. For more information on *playwav*, refer to the online documentation.

Checking the Version of Nuance

If you ever wonder which version of the Nuance System you have, simply type:

```
% nuance-version
```

This utility prints the version of the system that is installed, as well as the release date.

Check the License

The *check-license* utility indicates whether the Nuance System is licensed to run on the local machine. If the local machine is licensed, *check-license* prints the hostid and the date on which the license will expire, for example,

```
% check-license
Permission granted.
Current date is:      Fri Jan  3 17:57:50 1997
License for machine: bishop
                    with hostid:      0x807a20da
                    will expire:      Thu Dec 31 23:59:59 2020
```

Converting Mulaw Files to SPHERE Files

Raw mulaw data must be converted to SPHERE format before Nuance System programs can play or recognize it. The system provides the utility *mulaw2wav* for this purpose. See the online documentation for details.

Redirecting Application Output

Use *redirect* to redirect the output of Nuance applications. This program is particularly useful on Windows platforms. The usage is:

```
redirect [-i newin] [-o newout] [-e newerr] command
```

Note that if *stderr* and *stdout* are redirected to the same file, there may be some unpredictable interleaving. This depends entirely on the kind of buffering used by the program whose output is being redirected. If you want your application's output to look correct when *stderr* and *stdout* are redirected to the same file, add the following two lines:

```
setvbuf(stdout, NULL, _IONBF, 0);
setvbuf(stderr, NULL, _IONBF, 0);
```

This tells the system not to buffer output to *stderr* or *stdout*.

Normalizing Audio Files

Nuance provides several programs you can use to normalize *.wav* files. Nuance recommends that you normalize audio files used as prompts.

<i>wavnorm</i>	Lets you normalize a single <i>.wav</i> file, specifying the RMS. The recommended value is 2200.
<i>wavdynanorm</i>	Lets you normalize a set of <i>.wav</i> files.
<i>wavrms</i>	Lets you determine the average energy level of speech in a set of <i>.wav</i> files.
<i>wavxform</i>	Lets you perform operations such as clipping and filtering on <i>.wav</i> files.

Enter any of these at the command line to see a list of usage options.

STANDARD GRAMMARS

The Nuance System provides several standard grammars that you can include in your application's main grammar file by using the `#include` directive. These grammars, which can be found in `$NUANCE/data/grammars`, handle the following things:

- Numbers
- Dates
- Times
- Money amounts
- Yes/no confirmation
- Enrollment
- Touch-tone keys

Each version of a standard grammar has a suffix indicating which Nuance version it was initially created in. So, for example, *date.grammar-v6* was introduced as part of the Nuance version 6.0 release. A grammar file without a suffix—for example, *date.grammar*—is always a link to the latest version. Currently, *date.grammar* is a link to *date.grammar-v6*.

If you always want the latest and greatest date grammar, you should include *date.grammar* in your application's grammar. When you upgrade to a version of Nuance with a new date grammar and recompile your package, you will automatically be using that new grammar. If you prefer not to have things change underneath you, you should include a grammar with a version suffix; for example, *date.grammar-v6*. This grammar will not be changed in later Nuance versions (except possibly for bug fixes).

- **Note:** This grammar-naming convention has changed since version 5.

Here is more detail on the six standard grammars:

- *number.grammar* handles numbers from 0 to 999999. It returns the numerical value of the number expression. It does not handle digit strings like "one two three". The grammar intended for external reference is *Number*.

-
- *date.grammar* handles a variety of ways of saying dates, including context-dependent expressions such as “tomorrow” or “next thursday”. It returns a structure—for example, [*<month may> <day 3> <year 1996>*]. The grammar intended for external reference is *Date*.
 - *time.grammar* handles a variety of ways of saying times. It returns an integer value between 0 and 2359. In an expression like “four o’clock”, the grammar is ambiguous; it will return both 400 and 1600. The grammar intended for external reference is *Time*.
 - *money.grammar* handles a variety of ways of expressing monetary amounts. It returns a structure with two integer features, *dollars* and *cents*. The grammar intended for external reference is *AMOUNT*.
 - *yesno.grammar* handles a variety of ways of saying *yes* or *no*. The grammars intended for external reference are *CONFIRMATION* and *LOOSE_CONFIRMATION*.
 - *enrollment.grammar* is explained in “Enrollment: Adding Words by Speaking Them” on page 282. The grammar intended for external reference is *EnrollmentGrammar*.
 - *key.grammar* handles sequences of touch-tone key presses. It is useful only in applications built with the Dialog Builder. Many grammars are intended for external reference; see the comments at the top of the grammar file.

19

SHORT TOPICS

HIDDEN MARKOV MODELS

A variety of core technologies is currently available in commercial speech recognition systems. Techniques such as *expert systems* and *dynamic time warping* (DTW) possess some desirable properties, but *hidden Markov model* (HMM)-based recognition is by far the most powerful approach. In particular, the structure of HMMs and the training algorithm used to determine the recognition parameters make the approach well suited to speaker-independent, continuous-speech and large-vocabulary applications. Naturally, the Nuance System uses state-of-the-art HMMs at the core of its recognition engine.

HMMs function primarily as *acoustic models* that provide a mapping from the sampled speech signal to a sequence of phonetic units. The system is hierarchical, however, and HMMs are also used to map the phonetic sequence to the word sequence representing the transcription. The goal of this short topic is to provide a brief overview of HMMs and their application in the Nuance System. Understanding this material, while good for impressing friends at parties, is not essential for building applications. It may, however, provide insight as to how an application can best work or what capabilities are possible with HMM technology.

What's in a Name?

Although the theory and mathematics of HMMs are quite interesting (even considered beautiful by some), a complete description of HMM-based speech recognition goes well beyond the scope of this manual. Instead, the hope is to provide a general feel for hidden Markov models—what they are, how they are used for speech recognition, and what makes them better than other technologies. The best place to start (or *initial state*) is with the name *hidden Markov model*.

Why Is an HMM a Model?

The term *model* indicates that the approach is to model the speech signal as opposed to simply constructing a sentence classifier. At a conceptual level, an HMM is a statistical model of the speech production process (i.e., given a sequence of words, it can be used to generate a speech signal). As well as being a production model, HMMs can also be used to compute the probability¹ that a given speech signal was generated from a particular sentence.

In speech recognition, an HMM is constructed for every sentence allowed by the grammar. When new speech data are to be recognized, the system computes the probability that the data were generated by each HMM. The recognition result is the sentence that corresponds to the HMM that, with highest probability, generated the speech data.

What Is a Markov Model?

Because of the time-varying nature of the speech signal, it is important for a recognition system to have a mechanism to deal with time (or—in this digital world—*sequences of events*). In HMMs, a stationary Markov chain² is used as this mechanism. The basic premise is that the probability of transitioning from one *state* to another depends only on the two states and not on any prior transitions. For example, consider the four-digit PIN 8624. When a Markov chain is used, the probability of this sequence is simply:

¹For those who desire a more precise statistical treatment, it is actually a *likelihood*.

²Named for the Russian mathematician A. A. Markov (1856-1922).

$$\text{prob}(\text{first digit is } 8) \times \text{prob}(8 \rightarrow 6) \times \text{prob}(6 \rightarrow 2) \times \text{prob}(2 \rightarrow 4)$$

where $\text{prob}(A)$ denotes the probability of event A .

The advantage of using a Markov chain for modeling sequences and time can be seen from the PIN example. Modeling of all possible four-digit PINs would require 10,000 parameters. By applying the Markov assumption, this number is reduced to 110 parameters (100 for the digit transition probabilities and 10 for the initial digit probabilities). The probabilities should reflect the task. For example, assume that 90% of the users of a system choose their years of birth as their PINs.³ The model probability $\text{prob}(1 \rightarrow 9)$ should be set very high; otherwise, more weight would be given to less likely PINs, and the recognition error rate would increase.

Nuance uses Markov models at many levels of the system, in particular:

- Sentences are broken down into words (that is, the grammar—see Chapter 1).
- Words are broken down into phones (that is, the dictionary—see Appendix A).
- Phonetic units are broken down into start, middle, and end states.

Each of these sequences is modeled with a Markov chain. At the lowest level, states are allowed to transition to themselves. This self-loop allows HMMs to model the duration of a sound, which is very important because of the high variability in speaking rate for different talkers.

What Is Hidden?

By using the Markov chain framework, a sentence can be decomposed into a sequence of phonetic states. If this sequence is known, then recognition is (nearly) trivial. However, when recognition is performed, only the acoustic waveform is *observed*; the phonetic sequence is *hidden*. It is this hidden sequence that we attempt to recognize.

Any possibility of recognizing the utterance requires a relationship between the observed signal and the hidden Markov chain state. This relationship is

³Nuance Communications does not recommend using this method for PIN selection.

modeled by assuming that each state produces an acoustic observation independent of the observations produced by other states. The model used in the state-to-sound mapping determines the type of HMM: *discrete density* or *continuous density*. In almost all cases, the more accurate continuous-density HMMs provide sufficient recognition speed with the Nuance System.

Discrete-Density HMM

In a discrete-density HMM, a process called *vector quantization* (VQ) is used to map each frame of the speech waveform to one of a set of symbols. The discrete symbols are then used as inputs to the HMM. The advantage of this approach is that each HMM state must model only a finite number of symbols. This typically results in faster HMM computation. The drawback with this approach is that the coarseness of the VQ process can cause errors. This type of model is available with some versions of the Nuance System as the VQ model sets.

Continuous-Density HMM

This type of HMM models the relationship between acoustic frames and states by using (typically) mixtures of Gaussians. While being much more accurate than discrete-density HMMs, continuous-density HMMs do require more computation. This type of model is available with the Nuance System as the *genone* (gen) and *phonetically tied mixture* (PTM) model sets. (The two model sets also differ in computational requirements versus recognition performance.)

Recognition with HMMs

As described earlier, the recognition result is the sentence that corresponds to the HMM that would generate the observed acoustic waveform with highest probability. This approach seems straightforward, but of course there is a catch. It is impossible (or at least very difficult) to explicitly evaluate each possible sentence. For example, an eight-digit sequence has 100 million possible sentences. To reduce this combinatorial effect, the recognition system makes use of the hierarchical HMM structure (i.e., sentence HMMs are a concatenation of word HMMs, word HMMs are a concatenation of phonetic unit HMMs, and so forth).

During recognition, the computation can be shared across hypothesized sentences—that is, the same computation is required for the first seven digits of “1-2-3-4-5-6-7-8” as “1-2-3-4-5-6-7-7”. In most cases, however, even the

sharing of computation is not sufficient for real-time recognition, and *pruning* (deleting unlikely hypotheses) is required. The method used by the Nuance System for *search* (that is, evaluating the likely hypotheses) is the *Viterbi beam search*. This algorithm is very efficient and, combined with other proprietary techniques, gives the Nuance System the capability to recognize vocabularies of more than 10,000 words uttered in millions of possible ways. The pruning process can lead to improved recognition times at the expense of increased recognition errors. The Nuance System has, however, been tuned to provide the maximum recognition performance in the minimum time.

What Next?

This description of HMMs is a high-level overview. If you want to learn more about HMMs, numerous papers and books cover the subject at a variety of mathematical skill levels. A good jumping-in point is "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition" by L. R. Rabiner, *Proceedings of the IEEE*, 77(2):257-286, February 1989. In addition, you can contact the engineering team at Nuance Communications about any HMM questions you may have.

ENDPOINTING

The ability to perform *endpointing* allows more computationally efficient use of a recognition server's resources. When endpointing is not performed, all samples received by the client process are sent to the recognition server. Because users often pause after they are prompted to speak, some silence samples are unnecessarily sent to the recognition server. This increases the load put on the recognition server and lengthens the delay before the recognition server can respond.

Endpointing is the process of determining the beginning and the end of speech within the incoming sample stream. After the beginning and the end of speech are located, the region of speech is extended by a predetermined amount in each direction. Samples begin flowing to the *recserver* as soon as the onset of speech is detected, and the flow continues until the end of speech is found. In this way, the *recserver* can actually begin to process the utterance while the person is still speaking.

The following parameters control the performance of the Nuance endpointer:

`ep.StartSeconds`

Specifies the duration of speech activity required to determine that speech has started. Most application developers do not adjust this parameter.

`ep.EndSeconds`

Specifies the duration of silence required to determine that speech has ended. This parameter may be adjusted, depending on the type of answer that the application expects. For example, for simple “yes/no” responses, the parameter should be set to a small value, on the order of 0.4 second, so that the response will be snappier. For account numbers where talkers are likely to pause between groups of digits, the parameter can be set to a longer value to prevent the user from being cut off while pausing. This parameter can be set at application initialization via the command line or a *nuance-resources* file, but may also need to be set from within a running application, depending on the active grammar.

`ep.AdditionalStartSilence`

Specifies the number of seconds of prespeech silence to include with the speech when it is recorded or recognized.

`ep.AdditionalEndSilence`

Specifies the number of seconds of postspeech silence to include with the speech when it is recorded or recognized.

Trouble Shooting

Beginning of the Talker's Speech Is Missing

The endpointer detects speech by observing a rise in audible energy in the incoming signal. Sometimes the word that the user says may not have enough energy in the beginning of the word to set off the endpointer. As a result, only the middle portion of the word is sent to the recognizer. For example, if the user says “Cincinnati” the endpointer may not detect speech activity until the user has said the “natti” portion of the word because the “Cinci” portion is not emphasized. For these cases, the parameter `ep.AdditionalStartSilence`

Noises Are Detected as Speech

can be set to a longer value than the default 0.30 second to capture the missed beginning syllables.

In noisy conditions, the endpointer may be set off accidentally by surrounding noise sources such as machinery, traffic, and other people talking in the room. In general, the endpointer module can deal with a constant level of noise, such as machine hums or wind noise. However, random loud noises can set off the endpointer. To reduce the likelihood of false detection, the parameter `ep.ThresholdSnr` can be set to a higher value than the default value of 10. The trade-off is that the user would then need to speak more loudly to be heard. In extremely noisy conditions, a push-to-talk scheme may be required. By using `RCStartRecognizing()` and `RCStopRecognizing()`, the application can bypass the endpointing function, specifying the exact start and end of the speech to be sent to the recognizer. An application might have the user hold a key down while speaking, and tie the key-down and key-up events to these two functions.

Additional Uses

The endpointer can be used for recording without recognition. The RC API offers the `RCRecord()` function for this purpose. For example, Nuance's waveform editor, *Xwavedit*, uses `RCRecord()` to record utterances delineated by silence. As with any application using the Nuance System's endpointing functionality, the endpointer can be controlled from the command line. For example:

```
% Xwavedit ep.EndSeconds=2.0
```

modifies *Xwavedit* to require 2 seconds of silence before deciding that a recording is complete.

BARGE-IN

The *barge-in* feature of the Nuance RecClient allows a user to start talking and be recognized while an outgoing prompt is playing. Barge-in offers the following benefits to a speech recognition application:

- *More natural interaction.* System prompts can be interrupted at any time. The user thus has the option of canceling a wrong action or quickly jumping to a selection.
- *Shorter session time.* Instead of waiting for the prompt to finish before speaking, the user can speak at any time, accelerating the interaction with the application.

One scenario where barge-in is useful is in balancing the needs of novice users and expert users. Long introductory prompts that are required by novice users are not necessary for expert users. With the barge-in feature, the expert user can interrupt the introductory prompt without waiting for prompts to finish. It provides the benefit of allowing users who are already familiar with the application to interrupt explanatory prompts and to move quickly through the interaction.

Another example of the benefit of the barge-in feature occurs during the description of a menu of items. Many applications present a list of options for the user to choose from. It is natural and convenient to allow the user to choose the desired action as soon as the action has been heard. For example, the system prompts

“For car service, please say service. For sales, please say sales. For parts and warranty information, please say warranty.”

A user who wants to reach the sales department can interrupt the prompt and say “sales” as soon as the keyword for the sales department has been played. The user thus does not need to remember the keyword until the long list of menu items has been played.

Why Is Barge-In Necessary?

When an application plays a prompt, a portion of the outgoing energy is reflected in the input channel as an *echo*. This effect is more pronounced with analog telephone lines, but still exists even with digital lines such as T1 since the overall telephone network itself provides a path for reflecting the prompt. Without the barge-in feature enabled, the echo of the prompt can appear as input from the user and incorrectly trigger the endpointer. In addition, if the user speaks at the time the prompt is played, the input signal from the user is

mixed with the echo from the prompt, and the resulting input will have lower recognition accuracy because of the interference from the echo.

The barge-in feature is designed to minimize the effect of the echo on the input signal. However, it cannot completely remove this effect. Thus, a waveform recorded when the prompt and the user input are both present would contain some residuals of the prompt. These residuals are noises that could lower recognition accuracy.

The barge-in feature increases both the memory and the computation load of the RecClient, except when the Dialogic Antares board is used. However, the computation cost is present only while the prompt is being played.

There may be cases when the ability to barge in is not appropriate. For example, the user may be asked to confirm a certain transaction. To make sure that the user has heard the whole prompt before confirmation, user barge-in can be prevented by waiting for the Nuance event `NUANCE_EVENT_PLAYBACK_DONE` before calling `RCRecognize()`, or by setting the parameter `client.AllowBargin` to `FALSE`.

Including Barge-In in Your Application

Several steps should be followed to ensure reliable barge-in:

1. Crop out periods of silence within prompt messages.

To avoid having widely varying dynamic range in the outgoing prompt, it is advisable to crop off periods of silence at the beginnings and ends of prompts.

2. Normalize waveforms

It is important that the outgoing prompts have a uniform level of loudness. Otherwise, a sudden loud portion of the prompt would result in the echo of the prompt setting off the endpointer and misrecognition. Run the program *wavnorm* with the following command:

```
% mkdir new_prompt_dir  
  
% wavnorm 2200 original_prompt_dir/*.wav new_prompt_dir
```


For individual files that need to be normalized, the program *wavdynanorm* should be called as the following:

```
% wavdynanorm -dynamic_compression_at 2200 old.wav
new.wav
```

The *new.wav* file contains a version of *old.wav* that has its dynamic range compressed.

3. Concatenate all outgoing prompts and play them as one buffer.

Each time your application calls `RCPlayFile()`, the barge-in module needs to perform alignment between the outgoing prompt file and the incoming signal. This alignment requires time and CPU resources that can be saved if you make fewer calls to `RCPlayFile()`. Therefore, the application should store the waveform filenames in a buffer, and then play all the waveforms at once by using the ability of `RCPlayFile()` to play multiple files—for example, if the application needs to play the following two waveform files: *today_is.wav* and *friday.wav*. Instead of using the following code to play out the two files individually

```
RecClient *rc;
NuanceStatus status;
status = RCPlayFile(rc, "today_is.wav");
if (status != NUANCE_OK) {
    /* report error */
}

status = RCWaitForEvent(rc, NUANCE_EVENT_PLAYBACK_DONE,
    10.0, NULL, 0);
if (status != NUANCE_OK) {
    /* report error */
}

status = RCPlayFile(rc, "friday.wav");
if (status != NUANCE_OK) {
    /* report error */
}

status = RCWaitForEvent(rc, NUANCE_EVENT_PLAYBACK_DONE,
    10.0, NULL, 0);
if (status != NUANCE_OK) {
    /* report error */
}
```


the application should concatenate all the outgoing prompts into one text string and play them together as follows:⁴

```
RecClient *rc;
NuanceStatus status;
RCPlayFile(rc, "today_is.wav friday.wav");
RCWaitForEvent(rc, NUANCE_EVENT_PLAYBACK_DONE, 10.0,
NULL, 0);
```

For specific details on RCPlayFile() and RCWaitForEvent(), see the online documentation.

4. The application should call RCRRecognize() or RCRecord() immediately after the prompt has started.

In an application without the barge-in feature, the application would wait for the event NUANCE_EVENT_PLAYBACK_DONE before calling RCRRecognize() or RCRecord(). However, with the barge-in feature enabled, the application should start listening immediately after the playback has started.

For example, in an application without the barge-in feature, the sequence of calls to RCPlayFile() and RCRRecognize() would be

```
RecClient *rc;
RCPlayFile(rc, "today_is.wav friday.wav");
RCWaitForEvent(rc, NUANCE_EVENT_PLAYBACK_DONE, 10.0,
NULL, 0);
RCRecognize(rc, grammar, timeout_secs);
```

With the barge-in feature enabled, the code should be

```
RecClient *rc;
RCPlayFile(rc, "today_is.wav friday.wav");
RCRecognize(rc, grammar, timeout_secs);
```

The timeout_secs argument is automatically extended by the length of the prompt—for example, if an original application without barge-in plays a 3-second prompt file, and then has a timeout_sec of 2 seconds. With barge-in, the timeout_sec does not need to be changed because the

⁴In this and all following code samples, error checking has been omitted for simplicity.

timeout clock does not start counting until the prompt has finished playing.

Parameters Associated with Barge-In

`client.AllowBargeIn`

When set to TRUE, barge-in is enabled.

`client.KillPlaybackOnBargeIn`

When set to TRUE, outgoing prompts are automatically terminated when barge-in is detected during playback.

`audio.BargeInSNR`

This determines the threshold SNR to be used by the endpointing module during the outgoing prompt. This threshold can be raised when the application has the problem of prompts that barge in on themselves.

Important Points

When you use barge-in, keep in mind the following points:

1. If the user speaks during the first 0.5 second of playback, the barge-in module cannot align immediately, and thus the cutoff of the prompt may not occur. This limitation is due to the problem of using audio devices that do not provide precise information on when playback or recording has started. The problem does not exist when the Dialogic Antares board is used.
2. Currently, once barge-in is turned on it is on at all times. The human-machine interface is more consistent if the user can barge in at any time. However, for an important transaction, the application designer may want to ensure that the user has heard the complete prompt before answering. For these interactions, the application should wait for Nuance event `NUANCE_EVENT_PLAYBACK_DONE` before calling `RCRecognize()` to ensure that the user cannot skip through the prompt.
3. The barge-in feature has been optimized for speech only. Prompts that consist of nonspeech sounds, such as musical passages, will result in lower recognition accuracy.

Trouble Shooting

Problems that might occur with barge-in can be solved as follows:

1. The prompt barges in on itself.

For telephone lines that generate a high level of echo, sometimes loud regions in the outgoing prompt can set off the endpointer and cut off the outgoing prompt. One solution is to try to make the amount of energy in the prompt more level by using *wavdynanorm* as explained above.

Another solution is to increase the value of `audio.BargeInSNR` above its default value of 10. The trade-off is that the user may need to speak more loudly to barge in.

2. The prompt does not cut off when the user barges in.

There will be a short delay between when the user speaks and when the prompt is cut off. When the user barges in near the end of a prompt, the prompt may play to completion since the user's incoming speech was not detected in time.

3. Recognition accuracy is decreased when the user barges in.

The amount of residual echo from the outgoing prompt is high. Try turning down the volume of the output by setting the parameter `audio.OutputVolume` to a lower value. For even better results, Nuance Communications recommends using a digital telephone interface such as a Dialogic T1 or E1 card.

CONFIDENCE SCORING

The ability of the Nuance System to generate *confidence scores* allows applications to deal with the user more naturally when the system is uncertain of what the user has said. For example, suppose that the system is expecting a "Sunday/Monday/Tuesday..." response and the user instead says, "What should I do now?". The recognizer may return the hypothesis "Wednesday" because it is the choice that sounds most like what the user has said. However, by using the confidence score generated along with the hypothesis, the application can judge whether to follow through with the action requested or

to request one more confirmation, depending on the magnitude of the confidence score.

Applications can behave more naturally by taking into account how much confidence can be placed in the recognition result. When the user says something that the system does not expect, the system can reject the sentence based on the low confidence score instead of performing an action that the user does not anticipate.

Usage

Upon receiving a recognition result from the server, the application can call the function:

```
NuanceStatus RecResultOverallConfidence (RecResult *rr,
                                         int index, int *conf);
```

in the same way as calling `RecResultTotalProbability()` and get a confidence score that ranges from 0 to 100. The higher the score, the greater the degree of confidence that the sentence is actually the hypothesized sentence instead of random babble. Therefore, for words that sound acoustically similar to the hypothesized sentence, the returned confidence score would still be high. For example, if the system is expecting a day of the week to be said, the confidence score for three different sentences from the user might be as follows:

Sentence	Confidence Score
"Monday"	75
"Monkey"	48
"Abracadabra"	31

In general, a score above 50 indicates that the system is more confident of the incoming sentence being the hypothesized sentence rather than its being a random collection of words.

By calling `RecResultOverallConfidence()` within the application, you can use different thresholds at different points of the application. For example, if the system is expecting a yes/no confirmation on a purchase, you can

decide that a false acceptance of “yes” is more serious than a false acceptance of “no”. Thus, the confidence score for a “yes” recognition would be required to pass a higher threshold than a “no” recognition result.

Such a degree of customization within an application is not mandatory. The Nuance System also provides a method to automatically reject results with low confidence scores. The Nuance parameter `rec.ConfidenceRejectionThreshold` can be used to set a global threshold. Whenever the confidence score of the recognition result falls below the threshold, the recognition result is replaced with an empty string, effectively nulling this recognition result.

Parameters

The following parameters control the behavior of the system regarding confidence score generation:

`rec.GenConfidence`

When set to TRUE, confidence scores are generated. (Default is TRUE.)

`rec.ConfidenceRejectionThreshold`

When the confidence score for the result is lower than the threshold, the system replaces the recognized hypothesis with an empty string. (Default is 45.)

`rec.ClipConfidenceScore`

This function forces confidence scores in an N-Best list to be monotonically decreasing. (Default is TRUE.)

`rec.SuppressNullResult`

When set to TRUE, the system will remove recognition results in the N-Best list that have become empty strings because of having confidence scores lower than the threshold. (Default is TRUE.)

Tuning the Confidence Score Threshold

For each application, you should collect two sets of test data to be used for tuning the confidence threshold. The *in-grammar* set contains utterances that

are within the set of possible sentences covered by the grammar. The *out-of-grammar* set contains sentences not covered by the grammar. For example, if the grammar is designed for dollar amounts, the utterance “two hundred and eleven dollars” would be in the grammar, while the utterance “I would like to pay the gas company” would be out of the grammar.

Once the two test sets are assembled, the program *batchrec* can be used to determine a good confidence score threshold. A new option has been added to *batchrec*. Now, when *batchrec* is run with the parameter `-print_confidence_scores` on the command line, the program generates a table listing the recognition and rejection rates at different confidence score thresholds.

Run *batchrec* with the following parameters in addition to the usual parameters:

```
batchrec -print_confidence_scores ....
```

The first column in the generated table shows the confidence score threshold that was used in determining whether a recognition result should be rejected. Recognition results with a score greater than or equal to the threshold are kept, and recognition results with confidence scores lower than the threshold are rejected. At lower confidence thresholds, no hypothesis is rejected and the system is getting the maximum correct classification. As the threshold is adjusted upward, the percentage of hypotheses that are rejected increases.

Tables 28 through 30 are examples of the kinds of tables you might generate to help you tune the confidence threshold.

Table 28 summarizes Tables 29 and 30, which are from the in-grammar set and the out-of-grammar set. The goal is to maximize correct recognition rate on the in-grammar set while minimizing incorrect recognition rate on both the in-grammar set and the out-of-grammar set. As observed from Tables 29 and 30, the threshold can be chosen within a range of reasonable values.

As the confidence score threshold is adjusted from 45 to 55, the NL correct recognition rate for the in-grammar set decreases slightly, while the out-of-grammar incorrect recognition rate drops from 58% to 37%. You can judge which threshold is better to use by considering the cost of falsely rejecting an in-grammar utterance versus the cost of falsely recognizing an out-of-grammar utterance.

For applications that do not have a test set available, the threshold value of 45 is recommended, as it works well across a wide variety of applications.

TABLE 28: SUMMARY OF RECOGNITION ACCURACY

Confidence Threshold	In-Grammar NL Transcription		Out-Of-Grammar Transcription
	Correct (%)	Incorrect (%)	Incorrect (%)
.			
:			
.			
45	98.50	0.67	58.31
46	98.50	0.67	55.49
47	98.50	0.67	53.92
48	98.50	0.67	51.41
49	98.17	0.67	50.16
50	98.17	0.67	47.96
51	98.00	0.67	46.39
52	97.83	0.67	44.83
53	97.83	0.67	42.95
54	97.67	0.67	40.75
55	97.50	0.67	36.99
.			
.			
.			

TABLE 29 : IN-GRAMMAR SET RECOGNITION ACCURACY USING VARIOUS CONFIDENCE THRESHOLDS

Confidence Threshold	Transcription			NL Transcription		
	Correct (%)	Incorrect (%)	Reject (%)	Correct (%)	Incorrect (%)	Reject (%)
30	87.50	12.50	0.00	99.33	0.67	0.00
31	87.50	12.50	0.00	99.33	0.67	0.00
32	87.33	12.50	0.17	99.17	0.67	0.17
33	87.33	12.50	0.17	99.17	0.67	0.17
34	87.33	12.50	0.17	99.17	0.67	0.17
35	87.33	12.50	0.17	99.17	0.67	0.17
36	87.33	12.50	0.17	99.17	0.67	0.17
37	87.17	12.50	0.33	99.00	0.67	0.33
38	87.17	12.17	0.67	98.67	0.67	0.67
39	87.17	12.17	0.67	98.67	0.67	0.67
40	87.17	12.17	0.67	98.67	0.67	0.67
41	87.17	12.17	0.67	98.67	0.67	0.67
42	87.17	12.17	0.67	98.67	0.67	0.67
43	87.17	12.17	0.67	98.67	0.67	0.67
44	87.17	12.00	0.83	98.50	0.67	0.83
45	87.17	12.00	0.83	98.50	0.67	0.83
46	87.17	12.00	0.83	98.50	0.67	0.83
47	87.17	12.00	0.83	98.50	0.67	0.83
48	87.17	12.00	0.83	98.50	0.67	0.83
49	87.00	11.83	1.17	98.17	0.67	1.17

TABLE 29 (CONTINUED): IN-GRAMMAR SET RECOGNITION ACCURACY USING VARIOUS CONFIDENCE THRESHOLDS

Confidence Threshold	Transcription			NL Transcription		
	Correct (%)	Incorrect (%)	Reject (%)	Correct (%)	Incorrect (%)	Reject (%)
50	87.00	11.83	1.17	98.17	0.67	1.17
51	87.00	11.67	1.33	98.00	0.67	1.33
52	87.00	11.50	1.50	97.83	0.67	1.50
53	87.00	11.50	1.50	97.83	0.67	1.50
54	87.00	11.33	1.67	97.67	0.67	1.67
55	86.83	11.33	1.83	97.50	0.67	1.83
56	86.83	11.33	1.83	97.50	0.67	1.83
57	86.67	11.33	2.00	97.33	0.67	2.00
58	86.67	11.17	2.17	97.17	0.67	2.17
59	86.17	11.17	2.67	96.67	0.67	2.67
60	86.00	11.17	2.83	96.50	0.67	2.83
61	85.83	11.00	3.17	96.17	0.67	3.17
62	85.67	10.83	3.50	95.83	0.67	3.50
63	85.33	10.83	3.83	95.50	0.67	3.83
64	85.00	10.67	4.33	95.00	0.67	4.33
.						
.						
.						

**TABLE 30 : OUT-OF-GRAMMAR SET RECOGNITION ACCURACY USING
VARIOUS CONFIDENCE THRESHOLDS**

Confidence Threshold	Transcription		
	Correct (%)	Incorrect (%)	Reject (%)
.			
.			
.			
11	0.00	100.00	0.00
12	0.00	100.00	0.00
13	0.00	100.00	0.00
14	0.00	99.69	0.31
15	0.00	99.06	0.94
16	0.00	98.75	1.25
17	0.00	98.12	1.88
18	0.00	97.81	2.19
19	0.00	97.81	2.19
20	0.00	97.81	2.19
21	0.00	96.55	3.45
22	0.00	95.61	4.39
23	0.00	95.30	4.70
24	0.00	94.36	5.64
25	0.00	94.04	5.96
26	0.00	93.42	6.58
27	0.00	92.16	7.84
28	0.00	91.85	8.15
29	0.00	90.28	9.72
30	0.00	89.34	10.66

**TABLE 30 (CONTINUED): OUT-OF-GRAMMAR SET RECOGNITION ACCURACY
USING VARIOUS CONFIDENCE THRESHOLDS**

Confidence Threshold	Transcription		
	Correct (%)	Incorrect (%)	Reject (%)
31	0.00	88.09	11.91
32	0.00	86.83	13.17
33	0.00	86.52	13.48
34	0.00	83.70	16.30
35	0.00	83.07	16.93
36	0.00	81.50	18.50
37	0.00	79.62	20.38
38	0.00	76.49	23.51
39	0.00	74.61	25.39
40	0.00	70.85	29.15
41	0.00	68.03	31.97
42	0.00	65.20	34.80
43	0.00	63.32	36.68
44	0.00	59.87	40.13
45	0.00	58.31	41.69
46	0.00	55.49	44.51
47	0.00	53.92	46.08
48	0.00	51.41	48.59
49	0.00	50.16	49.84
50	0.00	47.96	52.04
51	0.00	46.39	53.61
52	0.00	44.83	55.17
53	0.00	42.95	57.05

TABLE 30 (CONTINUED): OUT-OF-GRAMMAR SET RECOGNITION ACCURACY USING VARIOUS CONFIDENCE THRESHOLDS

Confidence Threshold	Transcription		
	Correct (%)	Incorrect (%)	Reject (%)
54	0.00	40.75	59.25
55	0.00	36.99	63.01
56	0.00	34.80	65.20
57	0.00	33.54	66.46
58	0.00	32.29	67.71
59	0.00	30.09	69.91
60	0.00	26.33	73.67
.			
.			
.			

N-BEST PROCESSING

In some applications it is desirable to have the recognition engine generate a set of possible recognition results instead of only the best single solution. This capability is offered by the Nuance System's *N-Best* recognition processing method, which provides a list of possible recognition results, ranked from highest to lowest likelihood.

N-Best processing runs at a speed similar to that of 1-best recognition. There is little extra overhead in memory use during actual recognition. You may, however, want to use separate packages for grammars you will perform N-Best processing with to avoid any potential differences in memory usage.

Nuance N-Best Parameters

N-Best processing is enabled and configured by using the following Nuance parameters:

```
rec.DoNBest=TRUE/FALSE (default FALSE)
```

To initialize N-Best processing for use during recognition, set to TRUE.

```
rec.NumNBest=<Integer> (default 10)
```

Use to specify the maximum number of entries to be returned in the N-Best list.

- Note: If N-Best processing will be used at any point in the application, the application must be initialized with `rec.DoNBest=TRUE`. Then, at necessary points in the application, you can use 1-best processing by setting `rec.NumNBest=1` during runtime.

Using N-Best Results

When you use N-Best processing, the recognition server processes each utterance in the same manner as with the normal 1-best processing; however, the whole N-Best list is returned in the `RecResult` structure instead of only the best one. You can obtain an N-Best list of results by using the following Nuance API calls (assume that `recresult` is the `RecResult` structure returned from the recognition):

- To find the number of results in the N-Best list:

```
status = RecResultNumAnswers(recresult, &n_results);
if (status != NUANCE_OK) { /* ERROR HANDLING */ }
```

- To get recognition string number *i*:

```
status = RecResultString(recresult, i, char_buffer,
    sizeof(char_buffer));
if (status != NUANCE_OK) { /* ERROR HANDLING */ }
```

- To get the recognition score of result number *i*:

```
status = RecResultTotalProbability(recresult, i,
    &prob);
if (status != NUANCE_OK) { /* ERROR HANDLING */ }
```

- To obtain the natural language result for result number *i*, first obtain an `NLResult` structure through a call to `NLInitializeResult()`, and then fill it out by calling

```
status = RecResultNLResult(recresult, i, nl_result)
if (status != NUANCE_OK) { /* ERROR HANDLING */ }
```

- N-Best processing with $N > 1$ is not recommended for use with enrollment grammars (see the description in "Enrollment: Adding Words by Speaking Them" on page 282).

In all other respects, N-Best processing is identical to 1-best processing.

- **Note:** The number of results returned in the N-Best list may be less than the number requested. This may happen for a variety of reasons, such as low grammar complexity (if there are not many alternative paths), or a very good acoustic match to a small section of the grammar. Thus, you should always check the number of results in the list before accessing individual elements.



CREATING APPLICATION-SPECIFIC DICTIONARIES

The Nuance Speech Recognition System represents each word in the recognition vocabulary as a sequence of phonetic symbols. These symbols provide the correspondence between a word in the dictionary and its pronunciation. The Nuance System provides phonetic pronunciations for more than 100,000 English words. Ideally, all the words you need for your application will be part of the main Nuance System dictionary file. However, some applications may require words or alternative pronunciations, such as proper names or special acronyms, that the Nuance System dictionary does not contain. In those instances, you will need to specify additional word pronunciations in a separate dictionary file.

After creating the grammar specification file and running *nuance-compile*, you may see an error message indicating that some words in the grammar are not part of the main Nuance System dictionary. In this case, *nuance-compile* will not be able to produce a complete recognition system package, but instead will output a file called *name.missing*. This file will contain a list of all the words appearing in your grammar specification file but not in the *nuance-compile* dictionary. You now have two options as to how to proceed. The first option is to automatically generate the pronunciations by selecting the *-autopron* option when using *nuance-compile*; see Chapter 1 for a description of how to use this technique. The second option is to edit this file with any text editor and turn it into a complete dictionary file by adding pronunciations for each of the missing words. After you prepare the complete dictionary file,

rename it as *name.dictionary* and rerun *nuance-compile*, which will automatically find and use the new dictionary file.

Manually generated pronunciations that are carefully created are generally more accurate than those automatically generated.

Dictionary File Format

The Nuance System uses phonetic symbols referred to as *phones*. Roughly speaking, there is a phone corresponding to each spoken sound. For spoken English, there are 40 to 50 different phones. The phones used by the Nuance System, typically referred to as the *phone set*, are listed in Table A-1. Because letters can be pronounced differently when in different words and different letters can represent the same sound, there is not a one-to-one correspondence between letters and phones. For example, the [k] sound and corresponding k phone occur for the *c* in *cat*, the *k* in *keep*, the *ck* in *tick*, the *ch* in *echo*, and the *q* in *Iraq*. Similarly, the letter *c* can be pronounced by using the s phone (as in *circle*) as well as the k phone. The mapping between the pronunciations of a word, specified in terms of a phone sequence, and the spelling is a necessary input to the recognizer.

The auxiliary dictionary file is used to specify additional word pronunciations to *nuance-compile*. Each line of the dictionary file contains a word name followed by its pronunciation. The first non-whitespace string on the line is the word whose pronunciation is being defined. The remaining non-whitespace strings, separated by (non-newline) whitespace, are the sequence of phones defining a pronunciation for that word.

For example, the following shows how the words *telegraph*, *resettlement*, and *recollections* can be specified in a dictionary file:

telegraph	t eh l ax g r ae f
resettlement	r iy s eh dx ax l m ax n t
resettlement	r ax s eh dx ax l m ax n t
recollections	r eh k ax l eh k sh ax n z

Notice that two pronunciations are shown for the word *resettlement*. The compiled recognition package will recognize both pronunciations.

TABLE 31 : PRONUNCIATION MAPPING WITH THE NUANCE SYSTEM PHONE SET

Phonetic Category	Phone	Example	Phone	Example
Vowels	aa	father	ae	cat
	ah	cut	ao	caught
	aw	couch	ax	the
	ay	side	eh	bet
	er	bird	ey	date
	ih	dimple	iy	fleet
	ow	show	oy	toy
	uh	book	uw	blue
Stops	b	ball	p	put
	d	dice	t	try
	g	gate	k	catch
	dx	butter		
Nasals	m	mile	ng	running
	n	nap		
Fricatives	v	voice	f	friend
	z	zebra	s	sit
	dh	them	th	path
	ch	church	jh	judge
	zh	vision	sh	shield
	hh	have		
Approximants	r	row	l	lame
	y	yes	w	win, which

TABLE 31 (CONTINUED): PRONUNCIATION MAPPING WITH THE NUANCE SYSTEM PHONE SET

Phonetic Category	Phone	Example	Phone	Example
Others	-	(silence)		
(reserved for	inh	(inhale)	exh	(exhale)
Compiler)	clk	(click)	rej	(other)

It is important to be very careful when using the phone set, because the symbols representing phones do not consistently correspond to the letters in the spelling of a word. Since one phone can be represented by several different letters or letter combinations, those same letters can represent other phones in other words. Table A-2 shows typical mappings from English letters to phones.

- ❖ Caution: In using the phone set, do not use the phones **inh**, **exh**, **clk**, and **rej**. They are used only by *nuance-compile*.

TABLE 32 : PRONUNCIATION MAPPING WITH ENGLISH-LANGUAGE LETTERS

English Letter	Phone	Phone Example
<i>a:</i>	ey	able
<i>a:</i>	aa	father
<i>a:</i>	ae	apple pan
<i>a:</i>	ao	ball, caught, pawn
<i>a:</i>	ax	alive, zebra (unstressed vowels)
<i>b:</i>	b	ball
<i>c:</i>	k	cat, echo
<i>c:</i>	ch	child
<i>d:</i>	d	dogs
<i>d:</i>	t	asked, kissed
<i>e:</i>	iy	even

TABLE 32 (CONTINUED): PRONUNCIATION MAPPING WITH ENGLISH-

English Letter	Phone	Phone Example
e:	eh	echo
e:	uw	new
e:	ax	agent
f:	f	father
g:	g	golf
h:	hh	hotel
i:	ay	ivory, time
i:	ih	India
i:	iy	Lisa, shield
i:	ax	sanity, aphid
j:	jh	jump
k:	k	key
l:	l	law
l:	ax l	able
m:	m	Mike, thumb
m:	ax m	Adam, wisdom
n:	n	new, knot
n:	ax n	widen, nation
n+g:	ng	ring
o:	ow	open, coat, show
o:	aa	olive
o:	aw	cow
o:	ao	golf, bought, dog
o:	uw	shoe, loop

TABLE 32 (CONTINUED): PRONUNCIATION MAPPING WITH ENGLISH-

English Letter	Phone	Phone Example
<i>o:</i>	uh	book, would
<i>o:</i>	ah	compass
<i>o:</i>	ax	proceed
<i>o:</i>	oy	toy
<i>p:</i>	p	pull
<i>p:</i>	f	aphid
<i>q:</i>	k w	quick
<i>q:</i>	k	Iraq
<i>r:</i>	r	ring
<i>r:</i>	er	bird, rider, grammar
<i>s:</i>	s	soup
<i>s:</i>	sh	shell
<i>s:</i>	z	dogs, wisdom
<i>s:</i>	zh	vision
<i>t:</i>	t	time
<i>t:</i>	dx	butter, sanity
<i>t:</i>	dh	them
<i>t:</i>	th	thin
<i>t:</i>	ch	question
<i>t:</i>	sh	nation
<i>u:</i>	ah	under, putt
<i>u:</i>	uh	put, would
<i>u:</i>	uw	lunar, rude
<i>u:</i>	y uw	usage, confuse

TABLE 32 (CONTINUED): PRONUNCIATION MAPPING WITH ENGLISH-

English Letter	Phone	Phone Example
u:	ax	focus
v:	v	very
w:	w	wisdom, which
x:	z	xenophobia
y:	y	yes
y:	iy	very
y:	ay	cry
z:	z	zebra

As a general guideline for generating dictionary entries, ignore the spelling and focus on how the word sounds. Break it down into its component phones, sound by sound. Write the codes for those phones in the correct sequence. To double check, try to reproduce the sound of the word by reading the symbols in sequence. For example, in the word *backwards*, the sounds (represented by spelling, not phones) are as follows:

b, a, ck, w, ar, d, s

The letters are sufficient to tell us what the component sounds are, only because we can still see what the word is, and we have learned how to pronounce it. To tell the Recognizer that the sounds we hear should be spelled b-a-c-k-w-a-r-d-s, we need to be more precise about the exact sounds.

Using the Nuance phone set, we see that the [b] sound is straightforward, and the phone symbol for it is b.

The next sound is a little harder. The letter *a* can represent many different sounds, even in this one word. In the English letter table, we see that the second sound in *backwards* is the same as the first sound in *apple*, and is represented by the symbol ae.

The third sound is represented in the spelling by two letters, *ck*, but the symbol for this phone is just k.

The fourth phone is represented by the symbol *w*.

The fifth phone is tricky. It is a syllabic *r*, meaning that it has blended with the vowel to make up the root of the syllable, and is therefore one sound. The symbol for this phone is *er*. (In a Boston accent, the *r* part of the phone would be dropped, and the phone would be *ax*.)

The symbol for the sixth phone is *d*, and the final phone, which is spelled with an *s*, is actually represented by the symbol *z*, which is what it sounds like. The dictionary entry for this word would look like this:

backwards b ae k w er d z
backwards b ae k w ax d z

The second pronunciation indicates a Boston accent.

Be especially careful with words borrowed from other languages, words that would be hard for a child to read, and all vowels. The letter *s* can correspond to many sounds, such as the phone *z* as it does in *busy* and *dogs*. Similarly, the letter *d* can correspond to the phone *t*, as in *asked*, *missed*, and *pushed*.

It is useful to use the *pronounce* program to look through the main *nuance-compile* dictionary for example words that have similar pronunciations. At the command line, type *pronounce* and a list of one or more words for which you would like to see pronunciations. See the online documentation for *pronounce* for more details.

Tables A-3 and A-4 may also help in determining how to specify pronunciation in your dictionary. As shown in Table A-3, a word can have multiple pronunciations. This will depend on how many dialects you want your system to recognize. List multiple pronunciations on separate lines. Table A-4 is a sample dictionary made up of the words in Table A-2.

TABLE 33: EXAMPLES OF WORDS WITH MULTIPLE PRONUNCIATIONS

Word	Phones	Pronunciation
caught	k ao t	(cawt)
caught	k aa t	(cot)
either	iy dh er	(ee-ther)
either	ay dh er	(eye-ther)

TABLE 33: EXAMPLES OF WORDS WITH MULTIPLE PRONUNCIATIONS

Word	Phones	Pronunciation
farm	f aa r m	
farm	f aa m	(with Boston accent)
sandwich	s ae n d w ax ch	(very careful pronunciation)
sandwich	s ae n w ax ch	(dropping the "d")
sandwich	s ae m w ax ch	(blending "n" with "w" to get "m")
sandwich	s ae m ax ch	(dropping "w" altogether)
which	w ih ch	
which	hh w ih ch	

TABLE 34 : SAMPLE ENGLISH-LETTER DICTIONARY

Word	Phones
able	ey b ax l
adam	ae d ax m
agent	ey jh ax n t
alive	ax l ay v
aphid	ey f ax d
apple	ae p ax l
asked	ae s k t
asked	ae s t
ball	b ao l
bird	b er d
book	b uh k
bought	b ao t
butter	b ah dx er
cat	k ae t

TABLE 34 (CONTINUED): SAMPLE ENGLISH-LETTER DICTIONARY

Word	Phones
caught	k ao t
caught	k aa t
child	ch ay l d
coat	k ow t
compass	k ah m p ax s
confuse	k ax n f y uw z
cow	k aw
cry	k r ay
dogs	d ao g z
echo	e k ow
even	iy v ax n
father	f aa dh er
focus	f ow k ax s
golf	g ao l f
grammar	g r ae m er
hotel	hh ow t eh l
hotel	hh ax t eh l
india	ih n d iy ax
iraq	ax r ae k
iraq	ih r ae k
iraq	ay r ae k
iraq	ih r aa k
iraq	ax r aa k
ivory	ay v er iy
jump	jh ah m p

TABLE 34 (CONTINUED): SAMPLE ENGLISH-LETTER DICTIONARY

Word	Phones
key	k iy
kissed	k ih s t
knot	n aa t
law	l ao
lisa	l iy s ax
loop	l uw p
lunar	l uw n er
mike	m ay k
nation	n ey sh ax n
new	n uw
new	n y uw
olive	aa l ax v
olive	ao l ax v
open	ow p ax n
pan	p ae n
pawn	p ao n
proceed	p r ax s iy d
proceed	p r ow s iy d
pull	p uh l
put	p uh t
putt	p ah t
question	k w eh s ch ax n
question	k w eh sh ch ax n
quick	k w ih k
rider	r ay d er

TABLE 34 (CONTINUED): SAMPLE ENGLISH-LETTER DICTIONARY

Word	Phones
ring	r ih ng
rude	r uw d
sanity	s ae n ax dx iy
shell	sh eh l
shield	sh iy l d
shield	sh iy ax l d
shoe	sh uw
show	sh ow
soup	s uw p
them	dh eh m
thin	th ih n
thumb	th ah m
time	t ay m
under	ah n d er
usage	y uw s ax jh
very	v eh r iy
vision	v ih zh ax n
which	w ih ch
widen	w ay d ax n
wisdom	w ih z d ax m
would	w uh d
xenophobia	z iy n ax f ow b iy ax
yes	y eh s
zebra	z iy b r ax



A SAMPLE APPLICATION

This sample application illustrates the use of the advanced features of the natural language system described in Chapter 5. The files listed below can also be found online: *number.grammar* and *date.grammar* are in *\$NUANCE/data/grammars*, while the others are in *\$NUANCE/sample-packages*.

banking2.grammar

```
#include <date.grammar>

#include <number.grammar>

; Expressions like "my savings account", "checking" etc.

Account ( ?[my the]
    [ savings           {return(savings)}
    checking            {return(checking)}
    (money market)     {return(money_market)}
    (credit card)      {return(credit_card)}
    [(i r a) ira]      {return(ira)}
    mortgage           {return(mortgage)}
    ]
    ? account
)

; Basic commands
```

```

Command      [ transfer      {<command-type transfer>}
               withdraw      {<command-type withdraw>}
               Balance        {<command-type balance>}
               Pay-bill       {<command-type pay_bill>}
               ]

; Ways to ask for a balance

Balance      [ (what is [my the] balance)
               (tell me [my the] balance)
               ]

; Ways to specify bill payment

Pay-bill (pay ?for ?the Bill ?bill)

; Types of bill that can be paid

Bill         [ phone          {<bill phone>}
               utilities      {<bill utilities>}
               (cable ?(t v)) {<bill cable>}
               ]

; The possible parameters of a banking command

Parameter [ (from Account:acct) {<source-account $acct>}
             (to Account:acct)  {<destination-account $acct>}
             (in Account:acct)  {<balance-account $acct>}
             (Number:amt [dollars bucks]) {<amount $amt>}
             (?on Date:date)    {<date $date>}
             (?on Day:day)      {<day $day>}
             ]

; This grammar illustrates the special variable $string

Day          [ sunday monday tuesday wednesday thursday
               friday saturday ]
               {return($string)}

; A sentence is a command followed by zero or more
parameters

.Sentence (Command *Parameter)

```

number.grammar

```

; This is a grammar that covers numbers from 0 to 9999.
; It returns the numerical value of the number.

; Handles digit strings (e.g., "two five",
; "three six four nine").

Number [Digit:n Not_digit_digit:n Digit_digit:n
       Three_digit:n Four_digit:n]
       {return($n)}

; The different ways of saying 0.

Zero [zero oh] {return(0)}

; The digits 1 through 9.

Non_zero [one           {return(1)}
          two           {return(2)}
          three         {return(3)}
          four          {return(4)}
          five          {return(5)}
          six           {return(6)}
          seven         {return(7)}
          eight         {return(8)}
          nine          {return(9)}
          ]

; The digits 0 through 9.

Digit [ Zero:n Non_zero:n ] {return($n)}

; 10.

Ten ten {return(10)}

; 20, 30, 40...90.

Twenty_to_ninety [twenty   {return(20)}
                  thirty   {return(30)}
                  forty    {return(40)}
                  fifty    {return(50)}
                  sixty    {return(60)}
                  seventy  {return(70)}
                  eighty   {return(80)}

```

```

                                ninety      {return(90)}
                                ]

; 11 through 19.

Teen    [eleven      {return(11)}
         twelve      {return(12)}
         thirteen     {return(13)}
         fourteen     {return(14)}
         fifteen      {return(15)}
         sixteen      {return(16)}
         seventeen    {return(17)}
         eighteen     {return(18)}
         nineteen     {return(19)}
        ]

; Expressions like "five four".

Digit_digit ( Digit:n1 Digit:n2 )
             {return(add(mul(10 $n1) $n2))}

; Expressions like "twenty five", "seventy two".
; Not: "twenty".

Twenty_some (Twenty_to_ninety:n1 Non_zero:n2)
             {return(add($n1 $n2))}

; Ways of expressing numbers between 10 and 99 other
; than digit-digit.

Not_digit_digit [ Ten:n Twenty_to_ninety:n Teen:n
                  Twenty_some:n
                ]
                {return($n)}

; Ways of expressing numbers between 1 and 99 other
; than digit-digit.

Up_to_two [ Not_digit_digit:n Non_zero:n ]
           {return($n)}

; 10, 20 through 99.

Two_digit [ Ten:n Twenty_to_ninety:n Teen:n
            Digit_digit:n Twenty_some:n
          ]
          {return($n)}

```

```

; 10 through 99.
Teen_or_twenty_to_ninety [Teen:n Twenty_to_ninety:n
                          Twenty_some:n
                          ]
    {return($n)}

; 100
Hundred_hundred {return(100)}

; "a hundred"
A_hundred ( a hundred ) {return(100)}

; "one hundred", "two hundred"... "nine hundred"
Some_hundred ( Non_zero:n1 Hundred:n2 )
    {return(mul($n1 $n2))}

; "hundred", "a hundred", "four hundred" etc.
Hundred_exp [ Hundred:n A_hundred:n Some_hundred:n ]
    {return($n)}

; "seven hundred and forty two" etc.
Hundred_and_some ( Hundred_exp:n1 ? and Up_to_two:n2 )
    {return(add($n1 $n2))}

; "five sixty three" etc.
Digit_two_digit ( Digit:n1 Two_digit:n2 )
    {return(add(mul($n1 100) $n2))}

; numbers that can appear after, e.g., "five thousand"
Up_to_three [ Hundred_exp:n Hundred_and_some:n
              Up_to_two:n
              ]
    {return($n)}

; 100 through 999
Three_digit [ Hundred_exp:n Hundred_and_some:n
              Digit_two_digit:n
              ]
    {return($n)}

```

```

; 1000

Thousand thousand {return(1000)}

; 1000, 2000 etc.

Some_thousand (Non_zero:n1 Thousand)
    {return(mul($n1 1000))}

; "thousand", "one thousand", "two thousand" etc.

Thousand_exp [ Thousand:n Some_thousand:n ]
    {return($n)}

; "three thousand two hundred and forty two" etc.

Thousand_and_some (Thousand_exp:n1 ? and Up_to_three:n2)
    {return(add($n1 $n2))}

; "fifty two sixty three" etc.

Two_digit_two_digit(Not_digit_digit:n1 Not_digit_digit:n2)
    {return(add(mul($n1 100) $n2))}

; "forty five oh six" etc.

Two_digit_oh_digit ( Not_digit_digit:n1 oh Non_zero:n2)
    {return(add(mul($n1 100) $n2))}

; "seventeen hundred", "seventy two hundred" etc.

Many_hundred ( Teen_or_twenty_to_ninety:n Hundred )
    {return(mul($n 100))}

; "fifteen hundred (and) forty five" etc.

Many_hundred_and_some ( Many_hundred:n1 ? and
                        Up_to_two:n2 )
    {return(add($n1 $n2))}

; "one two three four" etc.

Four_digits ( Digit:n1 Digit:n2 Digit:n3 Digit:n4 )
    {return(add(mul($n1 1000) add(mul($n2 100)
                                add(mul($n3 10) $n4))))}

; 1000 through 9999

```



```

Four_digit [ Thousand_exp:n
              Thousand_and_some:n
              Two_digit_two_digit:n
              Two_digit_oh_digit:n
              Many_hundred:n
              Many_hundred_and_some:n
              Four_digits:n
            ]
{return($n)}

```

date.grammar

```

; This grammar file handles dates. The top-level grammar is
; Date. A feature structure with three features (month, day
; and year) is returned.

```

```

; No provision is made for context-dependent expressions
; like "today".

```

```

; In some cases, the structure returned will have fewer
; than three features. For example, if the input is
; "january first" or "the first".

```

```

Date [
  (?the DayOrd:d of Month:m ? Year:y)
    ;"the first of january (1995)"
  (Month:m ? the DayOrd:d ? Year:y)
    ;"january (the) first (1995)"
  (Month:m DayCard:n ? Year:y)
    ;"january one (1995)"
  (the DayOrd:d) ;"the first"
  (MonthNum:m DayCard:n ? Year:y) ;"one one (1995)"
]
{return([<month $m> <day $d> <year $y>])}

```

```

Month [ january      {return(january)}
        february     {return(february)}
        march        {return(march)}
        april        {return(april)}
        may          {return(may)}
        june         {return(june)}
        july         {return(july)}
        august       {return(august)}
        september    {return(september)}

```

```

        october      {return(october)}
        november     {return(november)}
        december     {return(december)}
    ]

    MonthNum [ one      {return(january)}
               two      {return(february)}
               three    {return(march)}
               four     {return(april)}
               five     {return(may)}
               six      {return(june)}
               seven    {return(july)}
               eight    {return(august)}
               nine     {return(september)}
               ten      {return(october)}
               eleven   {return(november)}
               twelve   {return(december)}
    ]

    DayOrd [ first     {return(1)}
            second     {return(2)}
            third      {return(3)}
            fourth     {return(4)}
            fifth      {return(5)}
            sixth      {return(6)}
            seventh    {return(7)}
            eighth     {return(8)}
            ninth      {return(9)}
            tenth      {return(10)}
            eleventh   {return(11)}
            twelfth    {return(12)}
            thirteenth {return(13)}
            fourteenth {return(14)}
            fifteenth  {return(15)}
            sixteenth  {return(16)}
            seventeenth {return(17)}
            eighteenth {return(18)}
            nineteenth {return(19)}
            twentieth  {return(20)}
            (twenty [first {return(21)}
                    second  {return(22)}
                    third   {return(23)}
                    fourth  {return(24)}
                    fifth   {return(25)}

```

```

        sixth      {return(26)}
        seventh    {return(27)}
        eighth     {return(28)}
        ninth      {return(29)}
    })
    thirtieth{return(30)}
    (thirty first){return(31)}
]

DayCard [ one      {return(1)}
        two        {return(2)}
        three      {return(3)}
        four       {return(4)}
        five       {return(5)}
        six        {return(6)}
        seven      {return(7)}
        eight      {return(8)}
        nine       {return(9)}
        ten        {return(10)}
        eleven     {return(11)}
        twelve     {return(12)}
        thirteen   {return(13)}
        fourteen   {return(14)}
        fifteen    {return(15)}
        sixteen    {return(16)}
        seventeen  {return(17)}
        eighteen   {return(18)}
        nineteen   {return(19)}
        twenty     {return(20)}
        (twenty [one      {return(21)}
                two        {return(22)}
                three      {return(23)}
                four       {return(24)}
                five       {return(25)}
                six        {return(26)}
                seven      {return(27)}
                eight      {return(28)}
                nine       {return(29)}
        ] )
        thirty     {return(30)}
        (thirty one){return(31)}
]

```

```

Year [ ( of this year )           {return(1995)}
      ( ? nineteen [ ninety      {return(1990)}
      (ninety Non_zero:n )       {return(add(1990 $n))} ] )
      ]

```

banking2.slot_definitions

```

command-type
source-account
destination-account
balance-account
amount
bill
date
day

```

banking2.slot_classes

```

; Note that the command-type slot is not included in these
; classes. This is because it is treated specially in the
; template definitions. See banking2.template_definitions.

```

```

; Slots that occur in almost every template

```

```

CommonSlot           (source-account amount)

```

```

; Slots that occur in the transfer template

```

```

TransferSlot         (CommonSlot destination-account)

```

```

; Slots that only occur in the bill payment template

```

```

OnlyInBillTemplate   (bill date day)

```

```

; Slots that occur in the bill payment template

```

```

BillSlot             (CommonSlot OnlyInBillTemplate)

```

```

; Slots that occur in the balance inquiry template

```

```

BalanceSlot          (balance-account)

```

banking2.template_definitions

```

; See banking2.slot_classes for the classes referenced
; here.

; The transfer template
(<command-type transfer> TransferSlot)

; The withdrawal template
(<command-type withdraw> CommonSlot)

; The balance inquiry template
(<command-type balance> BalanceSlot)

; The bill payment template
(<command-type pay_bill> OnlyInBillTemplate CommonSlot)

```

banking2.slot_combinations

```

; This rule says that if the command-type slot is filled ;
; with "balance", then the amount slot must not be filled.
; The rule is redundant given the template definitions in
; banking2.template_definitions.

IF <command-type balance> THEN NOT amount

; This rule says that if any slot in the OnlyInBillTemplate
; is filled, then the destination-account slot must not be
; filled.
; The rule is redundant given the template definitions in
; banking2.template_definitions.

IF OnlyInBillTemplate THEN NOT destination-account

; This rule says that if any slot in the OnlyInBillTemplate
; class is filled, then the command-type slot must be
; filled with "bill".

IF OnlyInBillTemplate THEN <command-type pay_bill>

; This rule says that if the destination-account slot is
; filled, then the source-account slot must also be filled.

IF destination-account THEN source-account

```

